

N° d'ordre : 1093

THESE

présentée en vue de l'obtention du grade de

Docteur de l'UNIVERSITE PAUL SABATIER de Toulouse

Spécialité : **INFORMATIQUE**

par

Rémi BASTIDE

OBJETS COOPERATIFS : UN FORMALISME POUR LA MODELISATION DES SYSTEMES CONCURRENTS

Soutenue le 6 Février 1992 devant la commission d'examen composée de :

MM	C.	GIRAULT	Président
	P.	COINTE	Rapporteur
	P.	SALLE	Rapporteur
	R.	VALETTE	Rapporteur
	M.F.	BARTHET	
	L.	FERAUD	
	J.C.	POUYTES	
	C.	SIBERTIN-BLANC	

Résumé

Cette thèse propose un nouveau formalisme, appelé Objets Coopératifs, dédié à la modélisation des systèmes concurrents, qu'il s'agisse d'ateliers flexibles de fabrication, de logiciel temps réel, de systèmes d'information, ou de toute autre système à événements discrets dans lequel la concurrence joue un rôle important.

Deux objectifs ont guidé la définition de ce formalisme : La recherche d'un grand pouvoir d'expression, permettant de construire des modèles clairs et concis, et la préservation du caractère formel, permettant l'exécution et l'analyse des modèles.

Le travail présenté se trouve au confluent de deux axes de recherche importants : d'une part les travaux portant sur l'expression de la concurrence au sein de l'approche Objet, et d'autre part les travaux relatifs aux Réseaux de Petri de Haut Niveau et aux techniques de structuration qui s'y appliquent.

L'approche Objet fournit les concepts qui permettent de structurer un modèle à l'image du système qu'il décrit : encapsulation données / traitements, classification, héritage, relation de type client / serveur. Les réseaux de Petri, quant à eux, permettent l'expression du comportement des objets, et de la dynamique de leur coopération.

Un Objet Coopératif est muni, comme dans un langage Orienté-Objet classique, d'une structure de donnée qui modélise son état, et d'un ensemble d'opérateurs (ou services) destinés à faire évoluer cet état. Il est de plus pourvu d'un comportement, modélisé par un Réseau de Petri à Objets. Ce comportement définit les règles d'évolution de l'objet, l'accessibilité des services qu'il offre en fonction de son état et, réciproquement, l'influence de l'exécution des services sur son état.

La communication entre objets s'effectue par invocation de services, suivant une architecture de type client / serveur. Le protocole de communication est lui même défini par un réseau de Petri. Il est donc possible, dans un système, de modéliser le comportement des objets et les communications qu'ils entretiennent de telle sorte que l'on puisse exprimer la concurrence aussi bien entre les différents objets qu'au sein du même objet.

Une classe d'Objets Coopératifs peut être décrite suivant deux points de vue. Sa spécification est destinée aux clients de cette classe, et définit le comportement ou mode d'emploi des objets instances. Son implémentation décrit le fonctionnement réel des instances, et notamment comment elles se comportent elles-mêmes en tant que clients d'autres objets. Des critères formels de cohérence entre spécification et implémentation sont définis.

Un algorithme est fourni, qui permet, à partir des réseaux de comportement des classes d'un système, de construire un réseau Prédicats / Transitions global pour tout le système. Cet algorithme prend en compte les problèmes relatifs à l'instanciation, à la relation d'utilisation dynamique, à l'héritage et au polymorphisme.

Le mémoire est organisé comme suit :

La première partie (chapitre I et II) est une étude bibliographique concernant les formalismes sur lesquels se fondent les Objets Coopératifs. L'approche Objet, et plus particulièrement la concurrence dans cette approche, est d'abord étudiée. On donne ensuite une définition détaillée des Réseaux de Petri à Objets, et on étudie les techniques de structuration des réseaux de haut niveau.

La deuxième partie est consacrée à la présentation du formalisme et constitue l'essentiel du travail présenté. Le chapitre III montre comment définir les classes d'Objets Coopératifs, aussi bien en spécification qu'en implémentation. Le chapitre IV traite de la définition d'un système d'Objets Coopératifs. Le chapitre V définit la sémantique formelle d'un tel système en donnant la technique de construction du réseau global. Le chapitre VI propose quelques extensions possibles au formalisme.

La troisième partie expose deux études de cas traitées au moyen de notre formalisme (Chapitre VII) : un atelier flexible de fabrication, et une interface personne / logiciel. On expose au chapitre VIII les techniques permettant d'exécuter un modèle, et le chapitre IX regroupe les différentes approches d'analyse formelle des propriétés qui peuvent être appliquées aux classes et aux systèmes d'Objets Coopératifs.

Introduction

"Le monde réel n'est ni plat ni séquentiel, mais au contraire multidimensionnel et hautement parallèle".

Grady Booch.

Le travail présenté dans ce mémoire se trouve au confluent de deux axes de recherche importants : d'une part les travaux portant sur l'expression de la concurrence au sein de l'approche Objet, et d'autre part les travaux relatifs aux Réseaux de Petri de Haut Niveau et aux techniques de structuration qui s'y appliquent.

Les concepts fondamentaux de l'approche Orientée-Objet sont aujourd'hui largement adoptés dans le domaine du développement de logiciel. L'un des intérêts majeurs de cette approche est que les abstractions du modèle restent proches des entités du système modélisé. Un modèle Orienté-Objet est alors plus simple à comprendre, à valider, à faire évoluer et à réutiliser que des modèles produits par des approches plus classiques.

La citation de G.Booch mise en exergue de cette introduction pose clairement le problème que rencontre un concepteur qui doit modéliser un système réel : les entités qu'il peut identifier sont souvent des entités actives (i.e. qui peuvent provoquer un changement de leur propre état), et dont le comportement n'est pas séquentiel. Si ce concepteur adopte une démarche Orientée-Objet, il doit chercher à transcrire dans son modèle l'aspect intrinsèquement concurrent des objets du réel, et selon [Sernadas 90], concevoir son système comme une "société d'objets évoluant concurremment qui coopèrent à la réalisation d'une tâche".

Les fondements mêmes de l'approche Objet (notamment l'encapsulation et la communication par messages) semblent bien adaptés à l'expression d'une concurrence et d'une répartition du contrôle entre les entités actives d'un système.

La difficulté majeure de cette approche réside dans la conception et la validation de la structure de contrôle qui résulte de cette coopération entre entités concurrentes.

Le but du travail présenté dans ce mémoire est de décrire un formalisme qui rende réaliste une telle approche, en intégrant aux concepts de l'approche objet un mécanisme permettant d'exprimer, à un haut niveau d'abstraction, les constructions fondamentales liées à la concurrence (synchronisation, parallélisme, concurrence des accès à l'information) et offrant des possibilités de validation formelle des modèles produits.

Les réseaux Place / Transition classiques rendent aisée l'expression des constructions relevant de la concurrence, et offrent de grandes possibilités d'analyse et de validation statiques ou dynamiques ; ils souffrent toutefois de la faiblesse de leur pouvoir d'expression, et de leur absence de prise en compte de l'interaction étroite entre données et contrôle au sein d'un système.

Des modèles de Réseaux de Petri de Haut Niveau (RdPHN), tels que les réseaux Prédicat / Transition ont été définis afin de résoudre certains de ces problèmes, et sont aujourd'hui largement utilisés.

Ces formalismes sont dévolus à la modélisation de systèmes concurrents dont on sait qu'ils atteignent rapidement une complexité très grande. Aussi les possibilités de structuration sont elles primordiales,

afin de permettre au concepteur de manipuler des modèles qui n'excèdent pas ses capacités de compréhension, et qui permettent la vision d'un modèle à différents niveaux d'abstraction.

Les techniques de décomposition fonctionnelle hiérarchique ont été les premières appliquées aux RdPHN et favorisent une approche descendante. Elles apportent les mêmes avantages, et souffrent des mêmes carences que lorsqu'elles sont appliquées à la conception de logiciel. Des techniques de composition de réseaux, favorisant une approche ascendante ont également été présentées, avec les mêmes buts.

L'objet de notre travail est de définir un modèle qui concilie les concepts de la conception Orientée Objet avec ceux des Réseaux de Petri de Haut Niveau (RdPHN), en conservant les avantages propres à chacune de ces approches, et notamment :

- Gérer la complexité des modèles, par des techniques et des critères sains de structuration ;
- Faciliter la réutilisation et l'extension de modèles déjà produits ;
- Offrir les mécanismes d'abstraction permettant de modéliser un système à différents niveaux, et d'utiliser le même formalisme à différentes étapes du cycle de vie ;
- Produire des modèles faciles à lire et à valider, en autorisant la description d'interfaces de haut niveau entre composants et en profitant de la représentation graphique claire et concise des Réseaux de Petri ;
- Permettre des validations statiques et dynamiques du modèle en cours d'élaboration, afin de détecter au plus tôt les erreurs et les dysfonctionnements, grâce à une sémantique formellement définie.

Pour atteindre ces objectifs, nous proposons un formalisme appelé **Objets Coopératifs**. Ce formalisme est défini formellement, avec toute la précision nécessaire pour qu'il puisse être effectivement utilisé comme un **Langage de Spécification Exécutable**. Le but est de mettre à la disposition du modélisateur les concepts structurant les plus puissants de l'approche objet (classification, encapsulation, héritage, composition) pour décrire les aspects structurels (ou statiques) d'un système, tout en profitant de l'expression claire, concise et formelle de la concurrence qui est une des caractéristiques de l'approche réseaux de Petri.

On peut envisager d'utiliser ce formalisme dans les domaines suivants :

- Le modèle peut servir de **spécification fonctionnelle** du système modélisé. On doit alors fournir un modèle qui se comporte comme le système, et dans cette optique les qualités primordiales d'un formalisme sont le pouvoir d'expression (un formalisme de haut niveau facilite la communication des modèles au sein d'une équipe) et le caractère formel (pour éviter toute interprétation subjective de la spécification) ;
- Le modèle peut servir à la **conception** du système. Les aspects structurels du modèle sont alors les plus importants, et les facilités d'abstraction et de structuration sont primordiales ;
- Le modèle peut servir à la **simulation** (ou au prototypage) du système. Le formalisme doit alors être suffisamment formel pour être exécutable ;

- Le modèle peut être **analysé**, afin d'établir certaines propriétés du système modélisé ;
- Le modèle peut permettre d'assister la **génération de code**, dans un langage informatique qui servira à l'implémentation du système final.

Le travail présenté ici s'attache principalement aux trois premiers points cités ci dessus : spécification, conception et simulation. Beaucoup reste à faire en ce qui concerne l'analyse et la validation d'un modèle d'Objets Coopératifs, mais le caractère formel de ce modèle permet d'envisager l'utilisation des résultats généraux de la théorie des RdP. Enfin nous n'avons pas étudié la génération de code à partir de notre modèle, mais des résultats disponibles par ailleurs [Paludetto 91] peuvent être utilisés.

La première partie du mémoire est consacrée à l'étude des deux formalismes qui servent de référence aux Objets Coopératifs : l'approche objet et les Réseaux de Petri.

- Le chapitre I est consacré à l'approche objet. Il rappelle de manière succincte les concepts fondamentaux de l'approche objet, qui sont maintenant bien connus (§I.1). Nous décrivons ensuite, de manière plus approfondie, les diverses propositions qui ont été faites pour prendre en compte la concurrence dans l'approche objet (§I.2). Enfin, le §I.3 présente une des méthodes les plus utilisées de conception par objets des systèmes concurrents : la méthode HOOD ;
- Le chapitre II décrit les réseaux de Petri à Objets (RPO), utilisés pour décrire le comportement et la coopération des Objets Coopératifs. La section II.1 est une présentation formelle et détaillée des RPO, nécessaire pour que nous puissions à notre tour définir formellement la sémantique d'un système d'Objets Coopératifs. En section II.2, nous proposons un certain nombre d'extensions originales aux RPO, que nous utiliserons au sein de notre formalisme. Les sections II.3 et II.4 sont une étude bibliographique des techniques proposées pour la structuration des modèles de RdP de Haut Niveau.

La deuxième partie est dévolue à la définition du formalisme des Objets Coopératifs, et constitue l'apport central de notre travail.

- Le chapitre III montre la manière de décrire une classe d'Objets Coopératifs, du point de vue de sa spécification aussi bien que de son implémentation ;
- Le chapitre IV décrit la constitution d'un système exécutable d'Objets Coopératifs, à partir de la définition des classes d'objets, et la manière d'organiser les classes en deux hiérarchies : la hiérarchie d'héritage et la hiérarchie de composition ;
- Le chapitre V est consacré à la sémantique formelle d'un système d'Objets Coopératifs. Nous montrons comment construire, à partir de la définition du système et de celle des classes qui le composent, un réseau de Petri unique qui définit la sémantique opérationnelle du formalisme ;
- Le chapitre VI propose des extensions au formalisme de base, afin de rendre plus facile la modélisation ou d'en étendre le domaine d'application.

La troisième partie présente deux études de cas, et traite ensuite des possibilités d'exécution et d'analyse statique d'un modèle.

- Le chapitre VII présente une étude portant sur la modélisation d'un atelier flexible de production (§VII.1) et une autre portant sur la conception d'une interface personne / logiciel (VII.2) ;
- Le chapitre VIII est consacré aux possibilités d'exécution répartie du modèle, et montre comment un système peut être exécuté au niveau de chaque objet, de chaque classe d'objets ou du système dans son ensemble ;
- Le chapitre IX décrit quelques unes des possibilités d'analyse statique des modèles, qui découlent de l'utilisation des RPO pour spécifier la dynamique des Objets Coopératifs.

PARTIE I :

OBJETS ET RESEAUX DE PETRI POUR LA MODELISATION DES SYSTEMES CONCURRENTS

Chapitre I. Modélisation par objets des systèmes concurrents

Une brique de verre servirait de cendrier. Une boîte ronde, en cuir noir, décorée d'arabesques à l'or fin, serait remplie de cigarettes. La lumière viendrait d'une vieille lampe de bureau, malaisément orientable, garnie d'un abat-jour d'opaline verte en forme de visière. De chaque côté de la table, se faisant presque face, il y aurait deux fauteuils de bois et de cuir, à hauts dossiers. Plus à gauche encore, le long du mur, une table étroite déborderait de livres.

Georges Perec : Les choses.

Près de vingt-cinq ans après les premiers efforts des pionniers [Dahl...66], l'approche par objets a pris une place prépondérante dans la pratique actuelle du génie logiciel. Les intuitions géniales des précurseurs de l'approche objet que furent Dahl et Nygaard ont subi une lente maturation, et forment aujourd'hui un corpus de concepts et de techniques suffisamment établi et structuré pour en faire un outil majeur de la qualité du logiciel.

Les concepts fondamentaux de l'approche objet se sont révélés suffisamment féconds pour être appliqués dans des domaines très différents de l'informatique. Parmi les applications les plus convaincantes des modèles à objets, on peut citer :

- La représentation des connaissances humaines ; dans ce domaine, l'accent est mis sur la flexibilité, la puissance d'expression et les possibilités d'accès associatif et d'auto-organisation des modèles [Minsky 75, 88] ;
- La modélisation des données, et plus spécialement les Bases de Données. L'important dans ce domaine est l'expressivité du modèle pour la modélisation de structures de données et de relations complexes, mais essentiellement statiques ;
- La construction de logiciel (aussi bien en conception qu'en développement), où les concepts d'objets sont mis en œuvre pour l'obtention d'une qualité supérieure en termes de fiabilité, de modularité, d'extensibilité et d'évolutivité.

Avec des objectifs aussi différents, les concepts mis en avant par chacune de ces approches sont souvent distincts, et parfois même contradictoires : ainsi une approche de type intelligence artificielle privilégiera des techniques souples favorisant une programmation de type exploratoire et incrémentale ; une approche de type génie logiciel, au contraire, utilisera des techniques plus contraignantes afin d'obtenir une fiabilité et une prouvabilité supérieure.

Le travail présenté dans ce mémoire se situe clairement dans le domaine de l'approche génie logiciel, aussi notre présentation de l'approche objet se limitera-t-elle à ce domaine.

Notre travail porte plus spécialement sur l'expression de la concurrence au sein de l'approche objet, aussi notre présentation des concepts de base de l'approche objet sera-t-elle brève, d'autant que les textes qui portent sur ce sujet abondent [Meyer 90b, Rumbaugh...91, Cointe 86, Booch 87], entre autres

Le chapitre est organisé comme suit :

- La section I.1 est un rappel des fondements de l'approche objet, en insistant plus particulièrement sur ceux qui ont guidé la conception de notre formalisme ;
- La section II.2 est une revue bibliographique assez détaillée des principales propositions faites pour intégrer la description de la concurrence et du parallélisme dans l'approche objet ;
- La section II.3, enfin, est consacrée à la méthode HOOD.

1. LES FONDEMENTS DE L'APPROCHE OBJET

Après un bref rappel des principes fondamentaux de l'approche objets, la section 1.1 rappelle la théorie des Types Abstraits de données, qui en est un des fondements théoriques.

La section suivante est consacrée à une étude sur les rapports entre spécification et implémentation des composants logiciels.

Nous présentons ensuite en détail deux concepts importants de l'approche objet que nous avons mis en avant dans le formalisme des Objets Coopératifs : la programmation par contrat et la hiérarchie d'héritage.

La programmation par objet est née du constat des faiblesses de la programmation structurée et de la conception par hiérarchie fonctionnelle descendante. Le problème majeur de ces approches est la dichotomie très nette entre structure des données et structure des traitements qu'elles induisent : le processus de conception s'attachant uniquement aux fonctions et à leur raffinement progressif, les données sont en quelque sorte diffuses dans toute la conception.

Or, au cours d'un processus de conception, on se rend compte que la structure que l'on envisage pour les traitements est bien plus souvent remise en cause que la structure des données. Il est donc plus judicieux de centrer le processus de conception sur les données (ou les «objets») manipulés. D'autre part, en centrant le processus de conception sur les données, la probabilité d'obtenir une certaine forme de réutilisabilité augmente, car dans de nombreux cas, un logiciel est une collection de structures de données plus ou moins «universelles» agencées pour obtenir une fonctionnalité «spécifique».

L'approche objet vise donc à permettre un processus de conception guidé par les données, mais s'attache à cacher la représentation interne de ces données en n'autorisant leur accès qu'à travers un ensemble d'opérateurs, qui masquent aux utilisateurs de l'objet les détails de son implantation physique. Ainsi la dichotomie entre données et traitement disparaît, et l'objet est une entité qui rassemble ses données propres et les opérateurs susceptibles de les manipuler.

[Rumbaugh...91] caractérise une approche orientée-objet par quatre critères :

- Le premier est l'**identité** : tout objet a une identité (sa **référence**) et les références d'objets sont uniformes et indépendantes du contenu, ce qui permet de constituer des collections d'objets hétérogènes ;
- Le second est la **classification** : tout objet est une instance d'une classe, qui définit sa structure et son comportement ;
- Le troisième est le **polymorphisme**, c'est à dire la propriété qu'un même traitement ait des interprétations différentes selon la classe de l'objet auquel on l'applique ;
- Le quatrième est l'héritage, qui permet de relier les différentes classes en une hiérarchie taxonomique, et qui est essentiel pour promouvoir l'extensibilité et la réutilisabilité des classes.

L'architecture d'un système d'objets est structurée suivant une organisation de type **client / serveur**. Un objet du système, qui communique avec les autres objets par envoi de messages, réclame

l'exécution d'un service à un autre objet, et en récupère le résultat éventuel. Nous utiliserons le terme d'**invocation** pour désigner l'action d'un objet client qui réclame l'exécution d'un service à un objet serveur.

Nous n'irons pas plus loin en ce qui concerne la présentation des concepts généraux de l'approche objet, mais ces différents points seront bien entendu repris en détail lorsqu'il s'agira de préciser comment ils s'intègrent dans notre formalisme.

Avant de poursuivre, prenons le risque de proposer une définition du concept d'objet. Cette définition ne prétend pas à l'universalité, mais sert seulement à caractériser les concepts sur lesquels notre approche se fonde.

Définition I.1 - Objet et Classe d'Objets.

Un objet est une entité qui possède un état, et qui offre à son environnement un ensemble d'opérateurs destinés à faire évoluer cet état.

Une **Classe d'Objets** définit "en intention" l'espace d'états des objets qui sont ses instances en décrivant pour ces objets une **structure de données**, un ensemble d'**opérateurs de changement d'état** et une **structure de contrôle**.

1.1. Les Types Abstrais de Données

Les Types Abstrais de Données (TAD) [Guttag 77, Liskov 77] sont une formalisation du concept d'objet dans une approche algébrique. Une description par Type Abstrait vise à obtenir une description complète, précise et sans ambiguïté d'une structure de données, sans pour autant se fonder sur la représentation physique de cette structure de données [Meyer 90b]. Un TAD est une abstraction d'un concept pour lequel une implémentation particulière n'est qu'une instance.

Un Type Abstrait de Donnée décrit une classe de structures de données en fournissant un ensemble de **services** et en précisant les **propriétés formelles** de ces services.

- Les services sont décrits par leur signature, c'est à dire leur espace de départ et d'arrivée. Ils donnent en quelque sorte la **syntaxe** du type ;
- Les propriétés formelles sont de deux ordres : les préconditions et les axiomes. Précondition et axiomes donnent à eux deux la **sémantique** du type.
 - Les préconditions définissent quand un service est applicable, en fonction de l'état de l'instance ;
 - Les axiomes décrivent les propriétés sémantiques invariantes des instances du type.

L'exécutabilité des Types Abstrais est assurée par des techniques de réécriture. L'aspect formel de cette approche de spécification permet d'effectuer des preuves de cohérence et de complétude des modèles produits.

La notion de type abstrait s'apparente à celle d'objet si l'on accepte d'associer à chaque champ un domaine de définition [Cointe 86]. C'est le choix qui est fait dans les langages à objets fortement typés tels que Eiffel [Meyer 90b] ou C++ [Stroustrup 87].

1.2. Spécification et implémentation des composants logiciels

La possibilité de distinguer la spécification d'un composant logiciel de son implémentation est un facteur clé pour la maîtrise de la complexité d'un système, et une des techniques de base préconisée par le génie logiciel. C'est aussi un des principaux apports de la théorie de Types Abstraits de Données à l'approche objet.

- La distinction entre spécification et implémentation favorise l'abstraction de données et le masquage d'information, et permet donc à l'utilisateur du composant de rester indépendant des changements qui pourraient intervenir dans l'implémentation de celui-ci ;
- Cette distinction permet d'envisager plusieurs implémentations possibles pour la même spécification, et ainsi de tester plusieurs alternatives de conception, sans pour autant remettre en cause l'architecture du système ;
- Elle permet de différer l'implémentation d'un composant sans entraver le développement d'autres parties du système, voire de confier cette implémentation à d'autres équipes de concepteurs, pour peu que la spécification en soit clairement définie ;

La distinction entre spécification et implémentation traduit les deux vues différentes que l'on souhaite avoir sur un composant du système.

a. Vue externe

La **vue externe** d'un composant est celle des utilisateurs potentiels de ce composant. Elle décrit ce que le composant fait et comment il peut être utilisé. Un composant est alors perçu comme une "boîte noire", dont on ne connaît que la fonction et le mode d'emploi. Cette approche bénéficie des fondements théoriques apportés par la théorie des Types de Donnée Abstraits. Fréquemment une analogie est faite entre cette vision de l'utilisation de composants logiciels et les techniques d'ingénierie que l'on rencontre dans le domaine de l'électronique, et que certains auteurs [Cox 86] ont proposé d'intégrer dans l'ingénierie logicielle avec le concept de circuits intégrés logiciels (software IC).

La spécification d'un composant logiciel, dont la connaissance est nécessaire pour son utilisation correcte, est constituée de deux composants distincts : l'**interface** et la **pragmatique**¹.

- L'interface décrit l'aspect **syntactique** de l'utilisation du composant ; si on poursuit l'analogie entre composant logiciel et circuit intégré, il s'agit là du brochage du circuit, de son nombre de pattes, des tensions acceptées en entrée et délivrées en sortie ;
- La pragmatique, au contraire, décrit l'aspect **sémantique** et **dynamique** du composant. La sémantique décrit les fonctionnalités, caractérise les résultats fournis en fonction des entrées et donne les effets de bord éventuels liés aux activités du composant. La dynamique décrit les enchaînements possibles d'actions sur le composant, et caractérise son évolution au cours du temps.

¹ Terme introduit dans la méthode TAXIS [Barron 82]

Les langages de programmation ou les méthodes de conception permettent à divers titres d'exprimer l'interface et la pragmatique lors de la spécification d'un composant logiciel :

- L'interface d'un composant logiciel est en général constituée de **services** (suivant les langages on parlera de procédures et fonctions, de méthodes, de routines,...) et d'**attributs** c'est à dire de données dont la valeur est gérée par le composant.
 - Un service est un opérateur destiné à faire évoluer l'état interne du composant, ou à obtenir une information sur cet état. L'interface définit, pour chaque service, son nom et sa **signature**, c'est à dire le nom et le type des paramètres acceptés, ainsi que leur mode de passage (par variable ou par valeur) et éventuellement le type de la valeur retournée par le service, si celui-ci est fonctionnel ;
 - Pour chaque attribut l'interface du composant définit son nom, son type et éventuellement les droits d'accès que des composants extérieurs ont sur cet attribut (lecture/écriture ou lecture seulement) ;
 - Certains langages permettent enfin d'inclure dans l'interface d'un composant des définitions de type, en général parce que ces types sont présents dans la signature des services, et seront donc nécessaires aux utilisateurs du composant. De même, certaines constantes symboliques peuvent être rendues publiques.
- La spécification doit en plus définir la **pragmatique** du composant ; par pragmatique, on entend à la fois son mode d'emploi, ses conditions d'utilisation, son comportement et le rôle qu'il peut jouer dans l'architecture d'un système. La pragmatique d'un composant inclut donc sa **fonction** et son **fonctionnement**. Ces différentes facettes, réunies, donnent la **sémantique** du composant qui est une information tout aussi importante que l'information syntaxique fournie par l'interface.

Il convient de noter que de nombreux langages ne permettent pas d'exprimer formellement la pragmatique dans la spécification d'un composant logiciel.

Ce que le langage Ada, par exemple, désigne sous le nom de spécification d'un paquetage n'est en réalité que son interface, et la pragmatique ne peut être fournie que par des inscriptions (commentaires, pseudo-code,...) qui sont extérieures au langage lui-même. A l'opposé, Eiffel exprime la pragmatique d'une classe en rendant publics des invariants associés à la classe et des préconditions et postconditions associées à l'exécution des services offerts, dans un esprit proche de la théorie de Types Abstraits de Données

b. Vue interne

La **vue interne** d'un composant décrit la manière dont ce composant est implémenté c'est à dire comment il est fait, et notamment comment il utilise d'autres composants du système pour remplir sa fonction.

La vue interne indique notamment les choix qui ont été faits pour la structure de données du composant et pour les algorithmes qui implémentent ses fonctions, ces choix devant autant que possible être cachés à ses utilisateurs.

Il est essentiel que l'implémentation d'un composant respecte sa spécification ; la pragmatique d'un composant est souvent spécifiée de manière informelle (e.g. en langage naturel) ou semi-formelle (e.g. par des formalismes graphiques [Ward...85, Harel...88, Buhr 84]), alors que l'implémentation est

limitée au cadre formel d'un langage de programmation. Le saut de niveau d'abstraction très important entre ces deux descriptions rend le plus souvent impossible toute vérification de conformité entre la spécification et l'implémentation de la pragmatique d'un composant.

1.3. Programmation par contrat

Le concept de **programmation par contrat** est un des apports les plus intéressants du langage Eiffel [Meyer 90b]. Les préconditions et les postconditions d'un service peuvent être considérées comme le contrat passé entre le service et ses clients.

- La précondition crée une obligation pour les clients. Elle définit les conditions que doit satisfaire un appel pour être légitime ;

```

Class File export
  enfile, défile, plein, vide, ...
feature
  enfile (x : INTEGER) is -- Ajoute x dans la file
    require
      not plein
    do ...
    ensure
      not vide
  end; -- enfile

  défile : INTEGER is -- Renvoie le plus ancien élément,
                      -- en l'enlevant de la file.
    require
      not vide
    do ...
    ensure
      not plein
  end; -- défile

```

Figure I.1 - Une File d'entiers en Eiffel

- La postcondition, en retour, crée une obligation pour le serveur : elle définit les conditions qui doivent être vérifiées au terme de l'exécution du service.

		Obligations	Bénéfices
enfiler	Client	S'assurer que la file n'est pas pleine	La file n'est plus vide (l'élément est inséré au début)
	Serveur	Insérer l'élément au début	Avant l'insertion, il reste de l'espace: nul besoin de traiter le cas où la file est pleine
défiler	Client	S'assurer que la file n'est pas vide	On récupère le plus ancien élément
	Serveur	Renvoyer l'élément le plus ancien	Avant l'opération, il y a quelque chose à renvoyer

Figure I.2 - Les deux aspects d'un contrat

L'énorme avantage de cette approche est de précisément qualifier les **responsabilités** au cours d'une invocation. Cette pratique permet d'éliminer les tests redondants qui sont une des sources de la complexité des logiciels, en s'assurant que les tests de validité des paramètres sont effectués une fois, et une seule.

L'exemple classique de la file permet d'illustrer les principes de la programmation par contrat. La Figure I.1 montre la spécification de la classe File, en ne donnant que les pré et post-conditions des services. La Figure I.2 résume les obligations respectives du serveur et du client lors des invocations

1.4. Hiérarchie d'héritage

Dans une démarche de modélisation, la définition d'une hiérarchie d'héritage entre classes a plusieurs significations :

- Tout d'abord, l'héritage permet de favoriser la **réutilisation** : en pratique, un système ne se construit pas "ex nihilo", mais au contraire s'appuie sur l'expérience acquise, et profite de développements antérieurs. En termes de conception orientée-objet, cela se traduit par le fait qu'on souhaite récupérer la conception d'une classe précédemment développée et simplement l'adapter à de nouveaux besoins, sans avoir à recommencer sa construction depuis le début. L'héritage est alors considéré comme un procédé essentiellement pratique permettant l'**extension** aisée des classes existantes. Dans cette approche, une classe est considérée comme un module, dont on souhaite récupérer et étendre les fonctionnalités ;
- L'héritage a également un sens plus fondamental, lié aux théories de représentation de la connaissance [Minsky 75]. Ici on cherche à structurer l'ensemble des classes en une taxonomie, permettant de grouper des entités conceptuellement proches. Lorsque l'on considère l'extension d'une classe, l'héritage porte alors une sémantique proche de l'inclusion ensembliste : Si une classe B hérite d'une classe A, alors l'ensemble des instances de B est inclus dans l'ensemble des instances de A. L'héritage décrit alors la **spécialisation** d'une classe par une autre; On dira alors qu'un B "est un" A (is-a) ou "est une sorte de" A (a kind of ou AKO) [Brachman 83, Cardelli...85]. La classe est alors considérée davantage comme un Type Abstrait de Données, et l'héritage formalise la notion de **sous-type**.

Dans la suite, nous adopterons le vocabulaire suivant : si B hérite de A, on dira que B est une classe **dérivée** ou **descendante** de A, et que A est un **ancêtre** de B.

a. Héritage simple

Les notions d'extension et de spécialisation sont souvent confondues dans les langages de programmation Orientés-Objet, et ne sont d'ailleurs pas totalement indépendantes. Toutefois, certains auteurs [America 87] proposent de trancher plus nettement entre l'héritage, considéré comme un concept lié à l'implémentation, et le sous-typage, qui construit une hiérarchie conceptuelle de classes proposant une interface de messages compatible.

America définit, pour une classe, une **interface d'utilisation** et une **interface d'héritage**, cette dernière constituant un ensemble cohérent d'attributs et de services. Dans le même esprit, [Horn 87] introduit la notion de **conformité** (conformance) entre classes, plus large que l'héritage, où deux

classes peuvent être conformes l'une à l'autre sans être explicitement reliées dans le graphe d'héritage, c'est à dire sans forcément partager le même ensemble d'attributs.

La vision conceptuelle de l'héritage (qui identifie classe et type) et sa vision pratique (qui identifie classe et module) peuvent parfois entrer en conflit : par exemple [Halbert 87], la vision pratique ferait de *Ellipse* une sous classe de *Cercle*, puisqu'une ellipse a les mêmes caractéristiques qu'un cercle, avec l'adjonction d'un centre supplémentaire ; ceci est bien entendu en contradiction avec la vision conceptuelle, où *Cercle* est une sous-classe de *Ellipse*, avec la contrainte que les deux centres soient confondus.

La notion d'héritage va rendre possible les constructions suivantes :

- La **surcharge** : Une classe dérivée peut fournir une nouvelle implémentation pour un service offert par un de ses ancêtres. Toutefois, si on désire conserver à l'héritage sa sémantique de sous-typage, la signature des services ne peut évoluer que dans le cadre de contraintes très strictes, formalisées dans [America 87] :

Définition I.2 - Sous-type

A' est un **sous type** de A si et seulement si :

pour tout service

$$s(p_1 : \text{type}_{A1}, \dots, p_i : \text{type}_{Ai}, \dots, p_n : \text{type}_{An}) : \text{type\_résultat}_A$$

exporté par A, A' possède un service de même nom :

$$s(p_1 : \text{type}_{A'1}, \dots, p_i : \text{type}_{A'i}, \dots, p_n : \text{type}_{A'n}) : \text{type\_résultat}_{A'}$$

tel que :

$$\forall i \in (1 \dots n) : \text{type}_{Ai} \text{ est un sous-type de } \text{type}_{A'i} ,$$

et : $\text{type\_résultat}_{A'}$ est un sous type de type_résultat_A

D'autre part, la surcharge d'une opération peut changer son implémentation mais doit conserver sa sémantique. Dans Eiffel, la sémantique d'une opération est exprimée par ses pré et post-conditions ; la préservation de la sémantique d'une opération au long de la hiérarchie d'héritage peut alors s'exprimer de manière formelle [Meyer 90] :

Définition I.3 - Préservation de la sémantique

Soit s un service exporté par la classe A et s' une surcharge de s dans A' descendante de A. $\text{Prés}'$ (la précondition de s') doit être plus faible ou égale à Prés_s , et sa postcondition $\text{Post}_{s'}$ doit être plus forte ou égale à celle de s, soit :

$$\begin{aligned} \text{Prés}_s &\Rightarrow \text{Prés}' \text{ et} \\ \text{Post}_{s'} &\Rightarrow \text{Post}_s \end{aligned} \quad (" \Rightarrow " \text{ désignant l'implication logique})$$

- Le **Polymorphisme** : une entité peut faire référence pendant l'exécution à des entités de classes différentes. On parlera alors, pour une entité, de son **type statique** (c'est à dire la classe par laquelle elle est définie) et de son **type dynamique** (c'est à dire la classe de l'instance qu'elle référence pendant l'exécution). Cette possibilité est contrainte par la construction suivante :
- La **Cohérence de type** : une entité de classe A ne peut référencer que des entités qui sont des occurrences des descendants de cette classe ;

- La **Liaison tardive** : La surcharge permet de fournir plusieurs implémentations du même service pour une hiérarchie d'héritage, et le polymorphisme autorise une entité à référencer des occurrences de différentes classes ; il est donc nécessaire d'avoir un mécanisme qui permette, pendant l'exécution, de choisir l'implémentation correcte du service à appeler en fonction de la classe de l'objet serveur. Ce mécanisme est connu sous le nom de liaison tardive ou dynamique (late or dynamic binding [Meyer 90b]).

La liaison tardive est souvent stigmatisée par les détracteurs de l'approche Orientée-Objet comme étant synonyme d'inefficacité. Il faut modérer cette affirmation, en constatant par exemple que, dans Eiffel, l'invocation, quoique légèrement plus coûteuse en temps qu'un appel de fonction, se fait en temps constant quel que soit la complexité du graphe d'héritage.

b. Héritage multiple :

On parle d'héritage multiple lorsque une classe peut avoir plusieurs antécédents immédiats ; le graphe d'héritage n'est alors plus un arbre, mais un treillis. Dans ce cadre, les deux aspects de l'héritage (vision d'une classe comme un module ou comme un type) sont présents :

- La puissance de représentation des connaissances est augmentée, car on peut dès lors regrouper au sein d'une même classe des caractéristiques de provenance diverse. Le danger potentiel de cette approche est de surcharger une classe de manière à ce qu'elle représente en définitive plus d'un type d'objet [Halbert 87] ;
- La réutilisation est facilitée : ainsi en Eiffel, l'héritage multiple est souvent utilisé pour émuler le mécanisme d'inclusion (`#include`) des langages modulaires tels que C ; on peut ainsi définir une classe comme une bibliothèque de fonctions utilitaires (par exemple des routines mathématiques). Si une classe a besoin d'une de ces fonction, il lui suffit d'ajouter la classe-bibliothèque à la liste de ses ancêtres.

Le seul problème posé par l'héritage multiple est la possibilité de conflit de nom entre des caractéristiques (services ou attributs) héritées de différents ancêtres. Les langages ou les formalismes supportant l'héritage multiple proposent divers mécanismes pour lever ces conflits [Ducourneau...87] : les techniques les plus fréquemment présentées sont la spécification d'un mode de parcours du graphe d'héritage [Stefik...86], qui conserve la première définition rencontrée de la caractéristique, ou le renommage explicite au niveau de la classe des caractéristiques en conflit [Meyer 90b].

2. MODELES D'OBJETS CONCURRENTS

Object languages differ, even in the fundamentals.

M.Stefik , D.G. Bobrow

Le rapport «Le Temps Réel» [CNRS 88], établi par le Groupe de Réflexion Temps Réel du CNRS, désigne "*La modélisation à objets des entités manipulées (événements et actions) et de leurs interactions*" comme un «Problème non résolu dans la spécification d'une application temps réel» .

Le problème de la Programmation Orientée-Objet Concurrente (COOP [Agha 90]) est cependant au cœur d'une quantité impressionnante de travaux de recherche, dont nous présentons ici un simple aperçu. La concurrence a d'ailleurs été, dès l'origine, associée au concept d'objet, puisque le LOO Simula, le premier, a introduit une forme de concurrence par coroutine.

Le concept d'objet, par son aspect faiblement couplé, encapsulé et autonome, semble bien se prêter à une implémentation distribuée, où chaque objet peut évoluer concurremment en échangeant des messages avec son environnement. Au delà de cette intuition de base, les approches proposées diffèrent non seulement par les solutions techniques ou syntaxiques adoptées, mais souvent aussi par les axiomes fondamentaux sur lesquelles elles se basent.

Nous avons tenté de classifier les différentes approches des LOO concurrents en trois grandes catégories, essentiellement en fonction de leur manière d'organiser "objets" ou "processus" à l'intérieur d'un système : les approches par moniteurs, par processus ou par acteurs. Les limites entre ces catégories sont bien entendu assez floues, et certaines approches ont des caractéristiques qui les placent à cheval sur ces frontières quelque peu schématiques. Nous espérons toutefois que cette classification puisse au moins servir de base de réflexion, car il nous apparaît que, dans chacune de ces catégories, les systèmes exhibent une topologie (statique) et une évolution (dynamique) assez caractéristiques.

Nous n'étudierons, dans chacune de ces catégories, que les aspects directement liés à la concurrence. Tous les langages examinés font en effet preuve, dans les autres domaines de l'approche objet (classification, héritage, instanciation, ...) d'une variété aussi grande que les langages à objets séquentiels.

2.1. Objets en tant que moniteurs

Dans un article corrosif, Leslie Lamport dresse l'histoire de ce qu'il appelle la concurrence *standard*, ce par quoi il entend l'état de l'art en matière de concurrence à différentes époques [Lamport 83] :

1965 :	Variables partagées
1968 :	Sémaphores
1972 :	Moniteurs
1978 :	CSP

Figure I.3 - Histoire de la concurrence selon Lamport

Il est remarquable de constater que les formulations initiales des Moniteurs (les "secrétaires" de [Dijkstra 71], puis [Brinch-Hansen 72, Hoare 74]), à peine postérieures aux premières publications sur Simula [Dahl 66], offrent déjà une indiscutable "saveur" Orientée-Objet. Il serait intéressant de retracer le cheminement des idées et des concepts de Dahl et Nygaard jusqu'à Hoare. Rappelons la définition des moniteurs [Hoare 74, Krakowiak 85]

Définition I.4 - Moniteur

Un **moniteur** est un mécanisme de structuration permettant de contrôler l'accès à des ressources communes. Il regroupe les **ressources partageables**, et les **procédures de gestion** de ces ressources.

Un moniteur est constitué par un ensemble de variables d'état et un ensemble de procédures qui manipulent ces variables. Certaines de ces procédures, dites externes, sont accessibles aux utilisateurs du moniteur, et constituent l'unique moyen de manipuler ses variables d'état. Un moniteur contient enfin un programme d'initialisation, exécuté une fois et une seule lors de la création du moniteur.

On voit que plusieurs des concepts principaux de l'approche Objet apparaissent dans cette définition : l'identité, l'encapsulation, la dissimulation d'information, l'instanciation dynamique.

L'aspect "encapsulation" est encore plus marqué lorsque l'on examine les primitives de synchronisation offertes par un moniteur : les conditions [Krakowiak 85].

Définition I.5 - Conditions d'un moniteur

Les synchronisations entre processus désireux d'accéder à la ressource s'expriment dans les procédures d'un moniteur au moyen de **conditions** : Une condition c est une entité du langage, qui ne peut être manipulée qu'au moyen de trois opérations, ou primitives :

$c.attendre$: bloque le processus p qui exécute l'opération, et le place «en attente de c ».

$c.vide$: fonction à valeur booléenne, vrai si aucun processus n'est «en attente de c ».

$c.signaler$: **si** $\neg(c.vide)$ **alors**
 <réveiller un des processus en attente de c >

fsi

Nous voici à nouveau en présence d'un Objet : la condition peut très bien être décrite comme un objet offrant les trois services cités ci-dessus, et qui encapsule une file des processus en attente. La condition est alors une version élaborée du **sémaphore** [Dijkstra 65]. Il nous paraît surprenant que le moniteur et la condition aient si rarement reçu une définition explicite en termes d'objets.

La motivation principale des moniteurs apparaît alors clairement : il s'agit d'un mécanisme structurant, qui permet d'encapsuler et de rationaliser l'utilisation des sémaphores, dont l'utilisation anarchique pouvait rendre la maintenance des systèmes concurrents extrêmement difficile, tout comme l'utilisation du GOTO rend illisible les programmes séquentiels.

La sémantique exécutoire des moniteurs est malheureusement rendue complexe en raison des axiomes relatifs à l'exécution des primitives :

- Les procédures d'un moniteur s'exécutent en exclusion mutuelle ; la conséquence implicite de cet axiome, peu souvent mentionnée, est qu'il doit exister une file d'attente, globale pour le moniteur, des processus en attente d'une procédure ;
- Un processus réveillé par *c.signaler* reprend son exécution à l'instruction qui suit immédiatement la primitive *c.attendre* qui l'a bloqué. La nécessité d'exclusion mutuelle entre les opérations d'un moniteur impose une contrainte supplémentaire sur le mécanisme de réveil : lorsqu'un processus *p* réveille un processus *q*, *p* et *q* ne peuvent être maintenus simultanément actifs. On impose donc à *p* de se bloquer, jusqu'à ce que *q* se bloque lui même, ou sorte du moniteur. Pour éviter le blocage indéfini de *p*, il faut en plus imposer que le transfert de contrôle de *p* à *q* soit ininterrompible, et garantir qu'un processus temporairement bloqué par l'opération *signaler* soit réveillé avant qu'un nouveau processus ne puisse exécuter une procédure du moniteur ;
- La dernière difficulté, enfin, est d'ordre méthodologique [Howard 76]. Le code d'une procédure d'un moniteur sera en général de la forme suivante:

```

si ¬Q /* Q : expression booléenne des variables d'état du moniteur
*/
    alors
        c.attendre
fsi
/* assertion : Ici Q est vrai */
< mise à jour des variables d'état >

```

Toutefois la forme syntaxique du moniteur n'impose pas que l'assertion soit effectivement vraie, puisque le réveil du processus par *c.signaler* est complètement dissocié de l'évaluation de la condition *Q*. Cette construction a donc pour inconvénient de faire apparaître la condition *Q* en deux points du programme, et de ne pas traiter simplement le cas où plusieurs processus attendent la même condition.

Kessels a proposé une primitive permettant de répondre à ces problèmes ; dans [Kessels 77] la primitive *attendre* est de la forme :

```

attendre(Q) /* Q : expression booléenne des variables d'état du
moniteur
*/

```

qui met le processus qui l'exécute en attente jusqu'à ce que *Q* devienne vraie ; dans cette approche, la procédure *signaler* n'existe donc plus. On verra par la suite que cette primitive a exactement la même sémantique que les *préconditions d'attente* de [Meyer 90a].

La discussion précédente sur les moniteurs, un peu longue, peut surprendre dans un rapport consacré à l'approche Objet ; l'approche par moniteurs, déjà ancienne, n'est-elle pas aujourd'hui périmée, supplantée par des concepts nouveaux, plus puissants et plus généraux ?

Il nous apparaît en fait qu'une partie importante des travaux visant à intégrer la concurrence dans l'approche Objet se réfèrent, parfois de manière implicite, aux concepts des moniteurs ; certaines approches consistent uniquement en une reformulation des concepts de moniteurs en termes d'objets, en y intégrant des idées plus nouvelles telles que l'héritage ou le polymorphisme.

Dans l'approche initiale de Hoare, le moniteur est une entité **passive**, à l'intersection des flots de contrôles initiés par les entités **actives** que sont les processus du système. La concurrence effective du système est liée au fait que les processus évoluent en parallèle, sauf aux points de synchronisation représentés par les entrées des moniteurs. On a donc une dichotomie très nette entre entités actives (les processus) et entités passives (les données partagées, encapsulées dans des moniteurs).

a. Trellis/Owl

Cette dichotomie se retrouve exactement dans l'approche proposée par J.E.B Moss, et implantée dans le LOO Trellis/Owl [Moss 87]. Le domaine d'application du langage est celui des systèmes de parallélisme à gros grain (large grain parallelism), par opposition au parallélisme à grain fin (implanté au niveau des instructions même du langage, ou par des architectures de type data-flow ou vectoriel). L'hypothèse de base est que *"la concurrence est exprimée par l'interaction de multiples activités sur des objets partagés"*.

Trellis/Owl définit un type spécial pour dénoter les activités du système. Il s'agit du type *Activity*. Une nouvelle activité peut être initiée par appel de la primitive *create* :

```
act := create(Activity, "FOO", {a1, a2, ...});
```

L'instruction ci-dessus crée une nouvelle activité qui exécute l'opération FOO avec les arguments a₁, a₂, ... Le flot de contrôle lié à l'activité persiste jusqu'à la fin de l'opération.

Le langage fournit les primitives de synchronisation nécessaires pour permettre l'accès aux données partagées :

- Les verrous d'exclusion (mutual exclusion locks), objets manipulés par les trois primitives *create*, *acquire* et *release*, permettent la définition de sections critiques ;
- Les files d'attente (wait queues) sont similaires aux conditions des moniteurs ;

Différentes primitives sont également fournies afin de répondre aux problèmes liés à l'utilisation des moniteurs (notamment pour permettre un réveil itératif des processus en attente dans une file).

Les approches suivantes restent dans la tradition des moniteurs, mais se dispensent de la notion de processus, en la confondant avec celle d'Objet.

b. CLIX, SINA

Dans le langage CLIX, proposé par [Hur 87], l'unique activité d'un objet est de répondre aux messages qui lui parviennent ; un objet n'a donc pas de "corps" (ou tâche de fond) et n'est actif que lorsqu'il traite un message.

Les objets peuvent communiquer :

- Par une primitive de type question / réponse : **ask**, pour lequel l'appelant demande l'évaluation d'un message, et est bloqué tant que l'appelé n'a pas fourni une réponse au moyen de la primitive **reply** ;

- Par une primitive de communication asynchrone : **send**, pour lequel l'appelant n'attend pas de réponse, et n'est donc pas bloqué ; l'appelant et l'appelé peuvent alors avoir une évolution concurrente.

On voit que seule la primitive **send** peut causer une évolution parallèle des objets dans le système, le couple **ask/reply** bloquant le flot de contrôle appelant et fonctionnant comme un appel de procédure distante.

L'originalité du langage SINA [Tripathi...89] est de permettre de modéliser à la fois la concurrence entre objets et la concurrence **interne** à un objet. La concurrence interne est implémentée soit par encapsulation de plusieurs objets actifs, soit par un mécanisme comparable au *fork* du système Unix, qui permet d'initier des flots de contrôle concurrent pendant l'exécution d'un service.

"Un objet SINA peut encapsuler une ressource partagée en même temps que ses mécanismes de synchronisation dans le même module. Ceci prend en compte plusieurs de problèmes posés par l'utilisation de structures de moniteurs emboîtés dans des systèmes qui gèrent des ressources structurées hiérarchiquement". Notons que, comme dans une approche classique par moniteurs, il peut exister des processus globaux au système, non encapsulés dans des objets.

c. Eiffel

Le langage Eiffel [Meyer 90a] ne contient pas de primitives permettant la concurrence (du moins dans sa version 3, commercialisée au jour où nous écrivons ces lignes). Toutefois, [Meyer 90b] décrit un modèle Orienté-Objet de programmation concurrente, et spécifie des extensions possibles du langage pour prendre en compte ce modèle.

En termes de concurrence, ses propositions peuvent se résumer ainsi :

- Les objets communiquent par invocation, avec une syntaxe d'appel identique à celle de Eiffel. L'unique mode de communication est donc l'appel de type question/réponse ;
- Un objet n'est actif que lorsqu'il traite un message, c'est à dire lorsqu'il est le destinataire d'une invocation.

Ces deux mécanismes, à eux seuls, interdisent à deux flots de contrôle d'évoluer concurremment au sein de deux objets différents, puisque le flot de contrôle appelant est toujours bloqué en attente de la réponse. Un mécanisme supplémentaire est donc nécessaire pour permettre à plusieurs flots de contrôle d'évoluer en parallèle :

- Il s'agit du mécanisme de l'**attente par nécessité** (lazy wait) dont le principe est le suivant :
Un flot de contrôle qui exécute une invocation de la forme `a := x.service` peut évoluer concurremment à l'exécution de `service`, et ce jusqu'à ce que la valeur de `a` soit effectivement nécessaire. La valeur liée à la variable `a` est effectivement nécessaire si `a` intervient dans une opération arithmétique telle que l'addition, dans une opération d'entrée sortie, ou est elle-même destinataire d'une invocation.

Un des grands intérêts de cette approche est de ne pas nécessiter de construction syntaxique spéciale pour dénoter une synchronisation, et de laisser le système lui-même synchroniser les flots de contrôle, en quelque sorte, au moment "le plus opportun".

Le point essentiel de la discussion de [Meyer 90a] n'est pas, toutefois, la description de ces primitives de communication, mais bien l'analyse des rapports entre la programmation concurrente et le modèle de la **programmation par contrat**, qui est un des fondements du langage Eiffel, et que nous avons présenté en §I.1.3.

Une constatation fondamentale est que "*Le modèle séquentiel du contrat ne tient plus tel quel pour la programmation concurrente*"; l'exemple de la file, détaillé au §II.2.7 suffit à s'en convaincre :

En mode séquentiel, un client du service *défiler* peut s'assurer qu'il remplit sa part du contrat (qui est de ne jamais réclamer ce service sur une file vide) par une construction de la forme :

```

if not ma_file.vide then
    x := ma_file.défiler
else
    ...
end

```

En mode concurrent, cette construction n'a plus de sens si l'objet dénoté par *ma_file* est **partagé** par d'autres objets, c'est à dire si d'autres objets possèdent eux aussi des références vers l'objet dénoté par *ma_file*. En effet, entre l'évaluation du prédicat *ma_file.vide* et l'appel *ma_file.défiler*, l'objet partagé peut servir un nombre quelconque d'invocations, pouvant notamment le rendre vide.

Meyer propose donc de modifier la sémantique des préconditions dans le cadre de la programmation concurrente : une précondition de la forme :

```

require not vide

```

s'interprète comme une **condition d'attente**, spécifiant ainsi que l'invocation est **bloquée** jusqu'à ce que la précondition devienne vraie (par suite de l'exécution par l'objet serveur de services réclamés par d'autres clients). Meyer souligne que "*les objets apparaissent alors plus proche des moniteurs que des processus*"; en effet, la sémantique des préconditions d'attente est exactement celle de la primitive d'attente proposée par [Kessels 77], en réponse aux problèmes soulevés par la primitive *signal* des moniteurs (cf §I.2.1).

Une des conséquences de ce nouveau modèle concurrent du contrat est que l'exécution d'un service n'est pas interruptible ; en effet, on rappelle que l'invariant d'une classe doit être vérifié avant et après l'exécution de chaque service, et que la pré- (resp. post-) condition de chaque service doit être vérifiée avant (resp. après) son exécution.

Au cours de l'exécution d'un service, invariant, pré- et post-conditions peuvent ne plus être vérifiés. Il faut donc empêcher l'interruption d'un service en cours d'exécution, car son contrat ne serait plus garanti.

Meyer propose de faire cohabiter les interprétations séquentielle et concurrente du contrat au sein d'un même système, en introduisant une notation particulière de déclaration des variables :

```

x : separate A

```

Cette notation définit que tout objet dénoté par x pendant l'exécution est susceptible d'être exécuté par un autre processeur (réel ou virtuel) que l'objet où cette déclaration apparaît. Lorsqu'on invoquera l'objet x , le client continuera à s'exécuter en parallèle avec l'exécution de sa requête. Toutes les préconditions des services appliqués à x doivent alors être interprétées comme des conditions d'attente.

Cette proposition nous paraît poser problème, pour la raison suivante :

- Le fait qu'une précondition doive être interprétée comme une condition d'attente n'est pas lié au fait que l'objet serveur soit actif (ou *separate*, dans la terminologie de Meyer), mais au fait qu'un objet soit **partagé entre deux ou plusieurs objets actifs** ; en voici un contre-exemple :

<pre>... a : INTEGER; f : File; x : separate Enfileur f.Create(...); x.Create(...); x.prends_une_file(f); a := f.défiler;</pre>	<pre>Class Enfileur export prends_une_file, ... feature prends_une_file(f : File) is do f.enfiler(3) end; ...</pre>
--	---

Figure I.4 - Contre-exemple

Dans le contre exemple de la Figure I.4, une file "passive" (non *separate*) est créée, puis passée en paramètre à un objet "actif" (*separate*), lui aussi nouvellement créé. La référence vers la file est donc partagée entre deux objets actifs. Selon Meyer, la précondition relative à l'appel *f.défiler* doit être interprétée en mode séquentiel, et donc provoquer une terminaison du programme si elle est testée sans être satisfaite. En réalité la précondition devrait être interprétée comme une condition d'attente, car l'objet de classe *Enfileur*, dénoté par x , va se charger d'entrer un nombre dans la file, débloquent ainsi l'objet qui l'a créé.

Le problème posé par ce contre-exemple pourrait être évité par une règle syntaxique forçant tous les paramètres passés à un client référencé par une variable *separate* à être eux-mêmes référencés par des variables *separate*, ou en effectuant alors un passage de paramètres **par copie** (la copie devrait alors être une **copie profonde**, c'est à dire prendre en comptes tous les objets liés au paramètre par la relation d'utilisation). Cette dernière solution est adoptée dans l'approche décrite par [Caromel 90], exposée ci-après.

Dans le formalisme qui fait l'objet de cette thèse, ces problèmes sont pris en compte de la manière suivante : toutes les entités sont conceptuellement *separate*, et un problème tel que celui du contre-exemple peut être résolu, car une analyse statique des classes du système peut permettre de détecter avant l'exécution ces cas de blocage. Ces points seront abondamment repris et commentés dans la deuxième partie du mémoire, dévolue à la description du formalisme.

Une dernière remarque enfin sur les propositions de [Meyer 90a] : la mention **separate** est associée aux variables du système, et non pas aux classes. Une même classe peut donc produire indifféremment des instances "actives" ou "passives", ce qui peut paraître surprenant, et va en tout cas à l'encontre de la plupart des approches présentées ici (voir par exemple l'analyse des propositions de [Caromel 90], plus loin dans ce chapitre).

d. Résumé

On peut caractériser une approche de type objet / moniteur par les propriétés suivantes :

- Un objet / moniteur n'a pas de flot de contrôle interne (tâche de fond, activité spontanée). Il n'est actif que lorsqu'il traite un message ;
- Un objet ne traite qu'un seul message à la fois ;
- Le plus souvent, il existe un mécanisme interne au langage permettant de mettre des "processus" en attente d'une condition (précondition d'attente de [Meyer 90a], files d'attentes dans le langage Trellis/Owl).

Dans une approche de ce type, la concurrence découle de trois mécanismes :

- L'existence de processus, entités actives qui déclenchent les méthodes des objets/moniteurs ;
- L'envoi d'un **signal** à un autre objet, sans attente de réponse. La réception du signal initie un flot de contrôle chez le receveur, qui peut évoluer parallèlement au flot de contrôle qui a émis le signal ;
- Le mécanisme de l'**attente par nécessité**, ou la réponse fournie par un appel de service n'est effectivement réclamée qu'en cas de besoin, ce qui permet au flot de contrôle appelant d'évoluer même sans avoir encore reçu la réponse de son appel.

2.2. Objets en tant que processus

[Meyer 90a] souligne les très fortes analogies que l'on ne peut manquer d'observer entre processus et objets :

- L'existence de données locales (les attributs d'un objet, les variables d'un processus) ;
- L'existence de données persistantes, gardant leur valeur entre les activations successives ;
- L'existence d'un comportement encapsulé (un cycle pour un processus, un ensemble de méthodes pour un objet).

En effet, un certain nombre de langages à objets cherchent à unifier la vision déclarative d'un objet comme instance d'un Type Abstrait de Donnée et sa vision procédurale en tant que processus séquentiel [Agha 90]. Parmi les langages qui se réclament explicitement de cette approche, on peut citer ABCL [Yonezawa...86], POOL [America 87], Concurrent Smalltalk [Yokote...87].

a. Ada

Nous ne souhaitons pas débattre ici du degré d'appartenance de Ada [Leverrand 82, Burns 85] à la famille des langages Orientés-Objet. Des discussions intéressantes sur ce sujet apparaissent dans [Stroustrup 87, Meyer 90b, BI-GL 49, BI-GL 57] entre autres.

Quoi qu'il en soit, le modèle de programmation concurrente promu par le langage Ada est bien connu, et a donné de nombreuses preuves de son efficacité.

Le modèle de base de communication est le rendez-vous, synchrone et asymétrique, où un client invoque une entrée de tâche et où la tâche appelée définit sous quelles conditions et dans quel ordre elle acceptera les requêtes sur ses entrées.

Il nous apparaît surprenant que le modèle des tâches Ada, simple et éprouvé, n'ait pas davantage trouvé sa place dans des propositions visant à introduire la concurrence dans l'approche Objet. L'idée d'un objet exécutant un corps de type boucle infinie, et spécifiant à divers points de cette boucle quels sont les messages qu'il accepte (avec éventuellement un indéterminisme sur le choix du message accepté) paraît pourtant intuitivement intéressante. Il faut par ailleurs remarquer que c'est la construction Ada du **type tâche** qui est le plus souvent utilisée lorsqu'on veut émuler en Ada les mécanismes des langages à objets [Pitette 86].

b. [Caromel 90]

L'intuition fondamentale qui sous-tend l'approche proposée dans [Caromel 90] est la poursuite de la démarche unificatrice promue par l'approche objet. L'approche objet a confondu les concepts de module et de type dans le concept unique de classe d'objets.

On souhaite aller plus loin en unifiant les concepts de processus et de classe d'Objets pour construire le concept de Classe d'Objets Concurrents (Figure I.5).



Figure I.5 - Unification des concepts de processus et de classe

Dans cette approche, l'**activité** d'un objet/processus consiste, après sa création, à exécuter son opération *Live*, qui décrit le **script**, ou le **corps** du processus. Un objet/processus peut donc être actif même sans être en train de répondre à une invocation.

La **communication** entre objets/processus prend la forme syntaxique d'un appel de méthode conventionnel (l'approche est décrite en utilisant la syntaxe du langage Eiffel). La sémantique d'exécution d'un tel service, toutefois, est relativement complexe.

L'exécution par un objet *o* d'une invocation de la forme :

p.méthode(paramètres)

déclenche les opérations suivantes :

- Levée d'une exception spécifique *Request*, dans le contexte de *p* ;
- Interruption de *p* ;
- Synchronisation entre l'appelé et l'appelant. Un objet de classe REQUEST est transmis de *o* à *p* ;
- *p* et *o* reprennent leur exécution en parallèle.

L'objet de classe REQUEST, transmis durant l'appel, spécifie la méthode à appliquer et les paramètres réels de l'appel. Le script de l'objet p va décrire à quel moment cet objet décidera de **servir** cet appel. L'article ne spécifie pas comment l'existence de variables dont le type est une méthode se combine avec l'aspect par ailleurs fortement typé du langage Eiffel.

Un des aspects intéressants de l'approche est la différence méthodologique qui est faite entre objets actifs (les processus, qui exécutent leur opération *Live*) et objets passifs, qui, comme leur nom l'indique, attendent passivement d'être invoqués. L'auteur souligne que le problème potentiel d'une telle approche est le partage d'une entité passive par plusieurs entités actives, qui donne lieu aux problèmes décrits au §I.2.c. La solution choisie pour résoudre ces problèmes est d'interdire totalement un tel partage, en se passant les paramètres qui dénotent des objets passifs **par copie**.

D'autres postulats qui sont à la base de l'approche nous semblent poser de sérieux problèmes.

- L'auteur présuppose qu'un processus qui n'est pas accessible (i.e. pour lequel aucun autre processus ne possède de référence) peut être terminé. Ceci semble faux :
 - Outre le cas trivial du processus-racine du système, qui ne doit pas être terminé abruptement, il faut considérer le cas d'un processus qui contrôle la terminaison correcte d'un ensemble d'autres processus, et qui ne doit pas être interrompu avant d'avoir achevé sa tâche ;
 - Un processus "non-accessible" peut cependant faire quelque chose d'utile, un exemple simple serait une horloge qui affiche l'heure en haut et à droite de l'écran ;
 - En toute généralité, il faut envisager et définir la sémantique du cas où le graphe de la relation d'utilisation entre processus comporte plusieurs racines, ou n'est pas connexe.
- De plus on suppose que le fait qu'aucun processus n'a de référence vers le processus p peut être reflété par un attribut (nommé *I_am_alone*) du processus p ; cette proposition, connue sous le nom de **comptage des référence** est décrite dans [Meyer 90b, pp 434-435], qui conclut que cette approche est totalement irréaliste, même dans un langage séquentiel, à cause de son coût : en effet chaque opération traitant des références, y compris une simple affectation de la forme $a := b$, modifie les compteurs de référence (dans ce cas le compteur de référence de l'objet précédemment associé à a doit être décrémenté, alors que celui de l'objet associé à b doit être incrémenté). Cette technique ne traite pas, par ailleurs, le cas des structures cycliques. L'utilisation d'un tel mécanisme dans un contexte distribué porte bien entendu ces problèmes à un ordre de magnitude supérieur, qui donnent lieu à de nombreuses recherches sur les approches de "ramasse-miettes" appliquées aux processus concurrents.

2.3. Objets en tant qu'acteurs

Le modèle des Acteurs pour la programmation concurrente a été, à l'origine, proposé par C.Hewitt et son équipe au MIT [Hewitt...73, Hewitt...77]. Les concepts qui sont à la base de ce modèle se retrouvent peu ou prou dans les deux approches décrites plus haut (l'approche objet / moniteur et l'approche objet / processus). Toutefois, le modèle des acteurs, qui a donné lieu à un grand nombre de travaux, a introduit des primitives idiosyncratiques qui méritent un chapitre à elles seules.

- Le clonage : dans l'approche par acteurs, le concept de classe est absent ; le mécanisme d'instanciation d'une classe, qui produit un nouvel objet, est remplacé par la notion de prototype [Senteni...90] : un prototype est à la fois un moule servant à la génération de nouveaux acteurs, et une instance dont les fonctionnalités sont étendues à la production de ses propres clones. Les clones sont définis de manière différentielle, en spécifiant uniquement ce qui les distingue de leur prototype. On peut de cette manière changer dynamiquement le comportement de toute une famille d'acteurs en changeant le comportement de leur prototype ;
- La communication : La seule primitive de communication entre acteurs est l'envoi de message, qui est *asynchrone* et *unidirectionnel*. Il n'existe pas, (du moins dans le modèle de base) de primitive de communication synchrone du type *question/réponse*. Ainsi, par exemple, l'accès par un acteur *x* à une variable d'état d'un autre acteur *y* nécessite-t-il deux transmissions de messages, avec un protocole bien défini :
 - *x* envoie une requête à *y*, réclamant la valeur de l'attribut ;
 - *x* se place ensuite en position de refuser tout message, sauf la réponse attendue de *y* ;
 - *y* transmet un message à *x*, contenant la valeur de l'attribut.
- La continuation : la réponse à un message n'est pas nécessairement retournée à l'expéditeur, mais à un acteur tiers, spécifié dans le message lui-même ; La continuation d'un message peut être l'émetteur lui-même, ce qui permet de considérer une communication du type question / réponse comme un cas particulier d'un mécanisme plus général.
- La délégation [Sallé 87] : Dans ce mécanisme, introduit par le langage Act-1 [Lieberman 81] tout acteur a connaissance d'un autre objet appelé son *proxy* (délégué ou mandataire). Si, au cours de l'exécution, un objet reçoit un message qu'il ne sait pas traiter, il fera suivre ce message à son mandataire, qui pourra à son tour le faire suivre à son propre *proxy*... Les langages d'acteurs n'ayant pas de notion de classe, de nombreux auteurs [Lieberman 86, Briot 87] ont proposé d'implémenter un mécanisme comparable au polymorphisme au moyen de la délégation, les appels se propageant le long de la chaîne des proxy de la même manière qu'ils remontent le long de la hiérarchie d'héritage, en recherchant l'implémentation de la méthode à exécuter ;
- La notion d'acteur de remplacement a été proposée dans les langages ACTORS [Agha 86] ou PLASMA II [Lapalme...89] pour remplacer le mécanisme de délégation. La notion d'affectation (ou de changement d'état) est alors remplacée par un mécanisme de plus haut niveau : le changement de comportement. Après traitement d'un message, un acteur remplace son comportement courant par un nouveau comportement, par la primitive *become*. En quelque sorte, en exécutant la primitive *become*, un acteur «pousse devant lui» un nouvel acteur qui commence instantanément à traiter les messages en attente, pendant que «l'ancienne version» de l'acteur continue le traitement du message initial, avant de disparaître.

a. ABCL

Un exemple caractéristique de langage fondé sur les concepts d'acteurs est le langage ABCL [Yonezawa...86]. Ce langage, toutefois, introduit un axiome fondamental qui n'existe pas dans le modèle d'origine des acteurs, et qui simplifie beaucoup sa sémantique d'exécution :

Définition I.6 - Axiome de préservation de l'ordre de transmission

Si un objet O envoie deux messages successifs à un objet T , la réception de ces messages par l'objet T doit être ordonnée de la même façon que leur émission.

Le langage ABCL qualifie également plusieurs protocoles de transmissions de message, que nous avons déjà rencontré ou que nous rencontrerons sous des formes voisines dans d'autres approches :

- Messages PAST : si un objet O envoie un message M de type PAST à un objet T , O continue son évolution dès l'envoi de M , sans attendre de réponse de la part de T . C'est le mode de communication unidirectionnel et asynchrone qui est à la base des modèles d'acteurs. On peut préciser le protocole des messages PAST, en spécifiant si O attend un **acquiescement** (accusé de réception) de la part de T avant de continuer, ou s'il continue sans acquiescement. On verra (§1.3) que le premier cas correspond aux requêtes LSER, et le deuxième aux requêtes ASER de la méthode HOOD ;
- Messages NOW : Dans ce protocole, O attend non seulement que T ait reçu le message, mais qu'il ait terminé le traitement associé et renvoyé le résultat. Ce protocole décrit donc une communication de type question/réponse, comparable à un appel de procédure distante. Il correspond aux requêtes HSER de HOOD ;
- Messages FUTURE : ce protocole reprend essentiellement la sémantique de l'attente par nécessité, décrite en §1.2.c.

Les objets ABCL sont **sérialisés** c'est à dire qu'ils traitent les messages dans leur ordre d'arrivée, un seul message à la fois. Il est prévu d'introduire dans le langage des objets non sérialisés, mais nous n'avons pas trouvé de description de ces objets dans les textes auxquels nous avons eu accès. En l'absence d'objets non sérialisés, le parallélisme est exprimé en ABCL de trois manières :

- Activation concurrente des objets, causée par les messages PAST ;
- Multi-casting (c'est à dire envoi simultané du même message à une famille d'objets) ;
- Parallélisme interne : le traitement associé à un message peut définir des flots de contrôle parallèle.

b. Résumé

Les modèles d'acteurs sont souvent caractérisés par une grande dynamique ; les acteurs sont créés en grand nombre, et souvent avec une durée de vie très courte (le calcul d'une factorielle par acteurs, par exemple, créerait un acteur pour gérer chaque étape de multiplication).

Cette grande dynamicité a conduit plusieurs auteurs à proposer des solutions pour maîtriser la complexité de la relation d'utilisation entre les acteurs :

- [Kaplan...89] décrit la relation d'utilisation entre les acteurs par un graphe, et utilise les **graph grammars** pour spécifier les changements possibles de la topologie de ce graphe ;
- [Engelfriet...90] décrit l'évolution d'un système d'acteurs par un réseau de Petri, qui spécifie les créations, les destructions, et l'évolution des références entre acteurs. L'analyse de ce réseau permet de détecter, par exemple, les cas de "référence pendante", ou un acteur garde une référence vers un autre acteur "mort".

[Stefik...86] souligne que les langages d'acteurs ont besoin des concepts structurants de classe et d'héritage : *"en pratique, ces langages disposent de «clichés» de programmation qui émulent les formes habituelles d'héritage empruntées aux langages à objet plus conventionnels"*.

3. LA METHODE HOOD

HOOD (Hierarchical Object Oriented Design) est une méthode de conception architecturale, conçue en premier lieu pour du logiciel destiné à être développé en Ada. Les fondements de la méthode HOOD se trouvent à la fois dans la méthode dite des **machines abstraites** (Abstract Machines ou AM) et dans la méthode OOD (Object Oriented Design [Booch 83]). Cette méthode, développée à l'initiative de L'Agence Spatiale Européenne (ESA) par un consortium constitué de Cisi-Ingénierie, CRI A/S et Matra Espace, est utilisée dans les projets Colombus et Hermès.

3.1. Les concepts

HOOD propose une décomposition hiérarchique de la conception dans l'espace de la solution, fondée sur l'identification d'abstractions dans l'espace du problème [HRM 89]. HOOD met l'accent sur la hiérarchisation de la solution, et impose deux types de hiérarchies :

- La hiérarchie **senior/junior**, où des objets seniors au sommet de la hiérarchie contrôlent et utilisent des objets juniors. C'est au sein de cette hiérarchie que sont prises en compte les contraintes de forte cohésion et de faible couplage ;
- La hiérarchie **parent/enfant**, qui permet la décomposition d'un objet en autres objets plus simples, et qui est considérée comme nécessaire pour permettre la sous-traitance dans le développement du logiciel.

Dans HOOD, l'unité fondamentale de modularité dans la conception est l'**objet**. Un objet est un modèle d'une entité du monde réel, qui combine les **données** et les **opérations** qui agissent sur ces données. Un objet possède une partie visible (**l'interface**), et une partie cachée (le **corps**, désigné en anglais par le terme **internals**) à laquelle les objets externes ne peuvent pas accéder.

- L'interface définit les opérations **fournies** et **requises** par l'objet, ainsi que les types, paramètres et exceptions associés ;
- Le corps constitue l'implémentation de l'interface fournie, en utilisant des opérations, des données ou des objets internes. Le corps modélise donc la structure interne de l'objet. Le corps d'un objet est décrit par les composantes suivantes :
 - Un **OpCS** (Operation Control Structure) pour chaque opération. L'OpCS décrit le flot de contrôle interne associé à l'exécution de l'opération, par exemple de manière algorithmique ;
 - Un **ObCS** (Object Control Structure) global à l'objet qui décrit les interactions possibles entre exécutions d'opérations. Dans la Figure I.6 [Vielcanet 89], cet ObCS correspond à la boîte centrale "Interactions de contrôle" ;
 - Les relations d'**inclusion** et d'**utilisation** que nous décrivons ci-après.

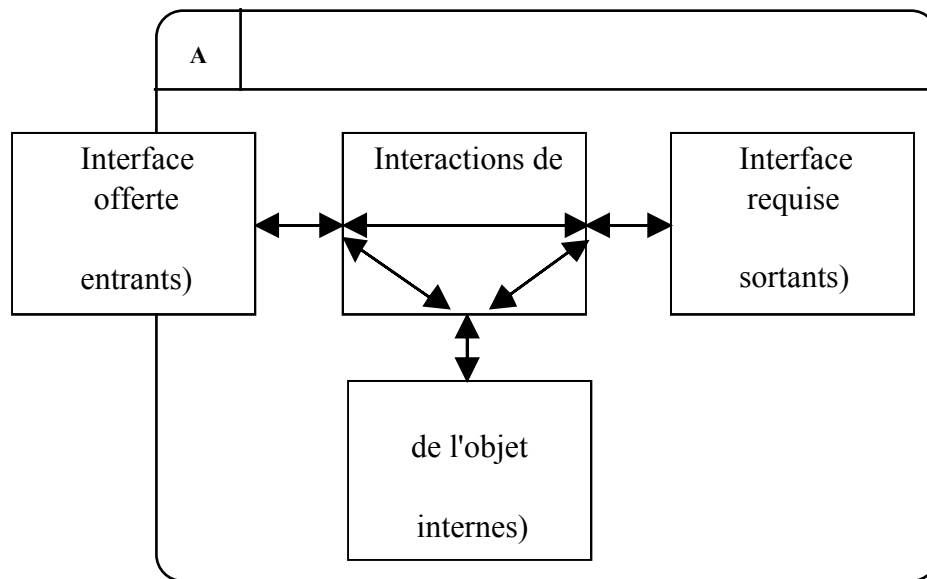


Figure I.6 - Dynamique des objets actifs HOOD

La communication entre un objet client et un objet serveur² est accomplie par l'exécution d'une opération appartenant à l'objet serveur. La sémantique de cette communication, définie par l'opération appelée, peut être définie de deux manières :

- Dans une opération **séquentielle**, le flot de contrôle appelant est transféré dans le contexte de l'opération appelée. L'OpCS associé définit alors la suite de l'évolution. Après achèvement de l'opération, le contrôle revient dans l'objet appelant. La sémantique d'une opération séquentielle est donc celle d'un appel de procédure ;
- Dans une opération **parallèle**, le contrôle n'est pas transféré, mais un flot de contrôle indépendant est créé dans l'ObCS de l'objet serveur, qui reçoit une **requête d'exécution** (execution request). La réponse à cette requête va dépendre de l'état du serveur. L'ObCS définit le flot de contrôle entre opérations dans l'objet client, alors que l'OpCS définit une logique interne séquentielle pour chaque opération. Une requête d'exécution peut donc **initier**, **terminer**, **interrompre** ou **réactiver** des flots de contrôle internes au serveur.

HOOD distingue deux types fondamentaux d'objets :

- Les **objets passifs** : un objet passif n'a pas d'ObCS et sa sémantique ne fait pas référence aux flots de contrôle. Lorsqu'une opération est invoquée sur un objet passif, le flot de contrôle appelant est transféré directement à cette opération, qui s'exécute séquentiellement. Les objets passifs n'ont donc aucune influence quant à la **structure de contrôle globale** du système ;

²HOOD n'utilise pas explicitement cette terminologie, mais parle d'objet utilisateur ou utilisé. Les concepts recouverts étant exactement les mêmes, nous conserverons donc la terminologie client/serveur.

- Les **objets actifs**, pour lesquels l'ObCS contrôle l'exécution des opérations fournies. L'exécution d'une opération peut donc dépendre de l'état de l'objet (qui a été modifié par des invocations précédentes). Si c'est le cas, on dit que l'opération a des **contraintes fonctionnelles d'activation**.

Lorsqu'il demande une opération à un serveur actif, le client peut contrôler l'effet de cet appel sur son propre flot de contrôle en spécifiant un **type de requête d'exécution** qui peuvent être :

- Requête HSER (Highly Synchronous Execution Request) : le flot de contrôle appelant est suspendu jusqu'à ce que l'opération invoquée ait été complètement exécutée ; ce mode d'exécution correspond exactement à la sémantique du rendez-vous en Ada ;
- Requête LSER (Loosely Synchronous Execution Request) : le flot de contrôle appelant est suspendu jusqu'à ce que la requête ait été acceptée par l'objet serveur. Le contrôle est rendu à l'objet client dès que le serveur a pris en compte la requête (envoi d'un acquittement) ;
- Requête ASER (Asynchronous Execution Request) : Le flot de contrôle appelant n'est pas interrompu, mais un signal d'exécution, ou "message" est envoyé à l'appelé. L'opération invoquée s'exécute alors en parallèle avec le flot de contrôle du client ;
- Requête TOER (Timed Out Execution Request) : l'appelant demande que la requête soit exécutée avant l'expiration d'un certain délai. Ce mode peut s'appliquer indifféremment aux HSER ou LSER.

HOOD ne spécifie pas de contrainte quant aux paramètres et valeurs de retour des opérations ; intuitivement, il semble toutefois évident qu'une invocation de type LSER ou ASER ne peut pas renvoyer de résultat.

3.2. La relation d'utilisation

La relation d'utilisation permet de définir le flot de contrôle entre les objets du système. Un objet A utilise un objet B si l'interface de A mentionne que celui ci requiert une opération fournie par B. HOOD définit un certain nombre de contraintes que doit respecter le graphe de la relation d'utilisation :

- Les objets actifs peuvent s'utiliser mutuellement sans contrainte ; le graphe de la relation d'utilisation entre objets peut notamment comporter des cycles, et le terme de hiérarchie d'utilisation est donc abusif en ce qui concerne les objets actifs. Les objets actifs peuvent également utiliser des objets passifs ;
- Les objets passifs ne peuvent utiliser que d'autres objets passifs, et le graphe de la relation d'utilisation entre objets passifs ne doit pas contenir de cycles.

Les principes de forte cohésion et de faible couplage [Parnas 73] sont vérifiables à l'examen du graphe de la relation d'utilisation : un objet doit utiliser aussi peu d'objets que possible, et être utilisé par le plus d'objets possible.

HOOD n'interdit pas à plusieurs objets actifs d'utiliser le même objet passif, mais ne spécifie pas non plus la sémantique d'une telle construction, qui correspond à une donnée partagée entre plusieurs processus.

3.3. La relation d'inclusion

La relation d'inclusion est le mécanisme proposé par HOOD afin de permettre une décomposition hiérarchique descendante (top-down) du problème. Un objet **parent** est alors décomposé en un ensemble d'objets **enfants** qui offrent collectivement la même fonctionnalité que le parent. Ce processus de décomposition peut alors être appliqué récursivement à chacun des enfants.

Une correspondance doit être donnée entre les opérations offertes par l'objet parent et celles offertes par ses enfants. Cette correspondance est forcément fonctionnelle ($1 \rightarrow 1$). Si une opération du parent est implémentée par plusieurs opérations des enfants (correspondance $1 \rightarrow n$), un objet enfant dédié (OP_Control object) doit être introduit pour assurer cette correspondance. L'objet OP_Control est un objet terminal, dont le corps se limite à l'OpCS de l'opération qu'il contrôle.

Un objet HOOD non terminal n'a pas d'élément qui lui soit propre, mais n'est constitué que par les déclarations de ses enfants et des indirections des services qu'il offre vers ceux-ci.

Les interactions entre flots de contrôle sont décrites dans l'ObCS de l'objet parent. Au niveau des enfants, cet ObCS peut être implémenté :

- Par un enfant dédié, forcément actif, et dont le nom est alors :
`Ctrl_<nom du parent>;`
- Par plusieurs enfants, dont l'un au moins sera actif.

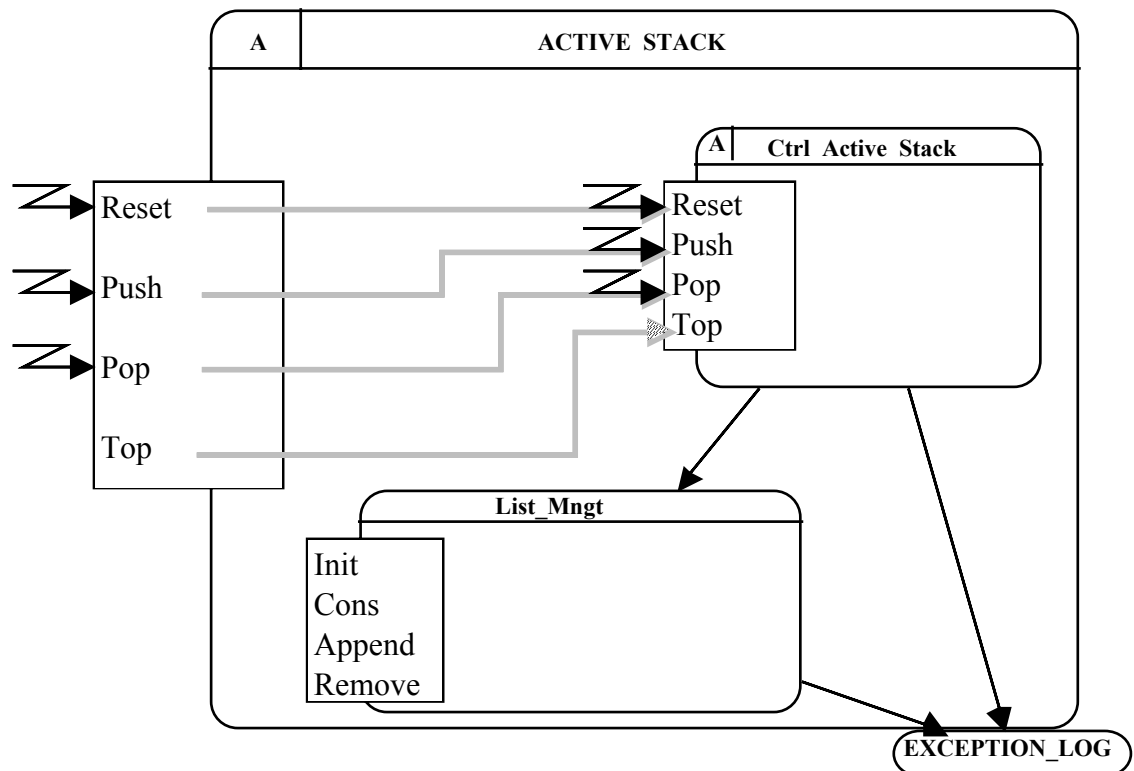


Figure I.7 - Syntaxe graphique des objets HOOD

La Figure I.7 [HRM 89] illustre la populaire représentation graphique des objets HOOD : la relation d'inclusion est matérialisée par le fait que les objets enfants sont dessinés à l'intérieur du parent. La relation d'utilisation entre objets frères est décrite par un arc de l'objet utilisateur à l'objet utilisé. La figure montre également quelques conventions graphiques de la méthode : les objets actifs sont désignés par la lettre **A**, à gauche de leur en-tête ; les opérations qui ont des contraintes fonctionnelles d'activation sont pointées par une flèche en zig-zag ; les flèches grisées entre opérations matérialisent le fait qu'une opération du parent est implémentée par une opération d'un enfant ; l'objet *Exception_log* est un objet "Oncle", c'est à dire un objet utilisé par *Active_stack* au niveau supérieur de décomposition.

3.4. Flots de données et d'exceptions

Les flots de données et d'exception sont mentionnés sur la représentation des objets par des annotations graphiques sur les flèches représentant la relation d'utilisation. Toutefois cette indication est assez peu lisible, car on ne sait pas quelles opérations sont concernées par ces échanges de données ou d'exceptions.

3.5. Les objets Classe

La notion de classe proposée par HOOD ne doit pas être confondue avec celle présente dans la majorité des autres modèles à objets. [HRM 89] définit une classe comme *"un objet réutilisable dans lequel certains types et / ou données utilisés par les opérations de la classe ne sont pas décrits. On peut définir une instance d'une classe dans laquelle ces types et/ou données sont définis explicitement, créant ainsi un objet particulier pour une conception donnée."*

Cette définition correspond tout à fait à ce qu'il est convenu d'appeler un **type générique** dans la plupart des autres méthodes de programmation.

Les limitations à l'utilisation des objets Classe sont sérieuses : un objet classe est forcément l'objet racine d'une autre conception HOOD ; il ne peut donc utiliser aucun objet du système à concevoir.

HOOD entretient la confusion avec les classes d'objet dans leur acception traditionnelle en proposant de plus une forme dégénérée de l'instanciation des langages à objet : quand plusieurs instances identiques sont nécessaires, le nom de ces instances est généré par un nom générique concaténé avec des entiers successifs pris entre deux bornes.

On voit que cette construction ne permet d'exprimer ni l'instanciation dynamique (où des objets viennent à l'existence au cours de la vie du système) ni la relation d'utilisation dynamique entre objets (où un objet client n'utilise pas toujours les mêmes objets serveurs).

3.6. Définition des ObCS

HOOD n'impose aucune notation particulière pour la définition des ObCS. L'environnement cible d'une conception HOOD étant le langage Ada, la manière la plus naturelle d'exprimer les structures de contrôle est d'utiliser les constructions présentes dans ce langage, à savoir les primitives de gestion des tâches et le rendez-vous. Toutefois d'autres formalismes sont proposés dans le manuel de référence HOOD : on peut citer le langage ESTEREL, les primitives de communication offertes par des exécutifs temps réels, ou les réseaux de Petri. La manière formelle d'intégrer ces formalismes externes n'est toutefois pas décrite.

3.7. Conclusion

L'**objet** est l'entité primordiale d'une conception HOOD, et toute la décomposition se fait en termes d'objets, et non pas en termes de classes, comme dans d'autres approches. On a vu (§I.3.5) que le concept de classe est présent en HOOD sous une forme assez éloignée de son acception usuelle dans les modèles à objets. Il est donc difficile en HOOD de factoriser la structure et le comportement d'objets identiques au sein d'une seule entité (la classe).

D'autre part des notions liées au concept de classe, telles que l'instanciation, l'héritage et le polymorphisme ne peuvent pas être prises en compte. Enfin, la relation d'utilisation (senior/junior) est définie statiquement par la conception entre des objets du système, ce qui interdit en pratique (ou rend très difficile) la définition de relations d'utilisation dynamiques, ou un objet client ne fait pas toujours référence aux mêmes objets serveurs au cours du temps.

Chapitre II. Les Réseaux de Petri de haut niveau

Trois grands modèles de la concurrence sont actuellement utilisés pour la description et l'analyse des systèmes parallèles : La logique temporelle [Audureau...87, 88, Pnueli 86], les approches relevant de l'algèbre des processus [Hoare 78, Milner 80, 83] et les Réseaux de Petri (RdP).

Une littérature très abondante existe sur les réseaux de Petri, leurs fondements théoriques, leurs applications pratiques et les divers dialectes de RdP de Haut Niveau, dérivés du modèle initial proposé par C.A.Petri [Petri 62]. Parmi la bibliographie «de base» sur les RdP citons notamment [Brams 83, Peterson 81, TSI 85] ainsi que les deux copieux tomes des "Lecture notes in Computer Science" [LNCS 87], actes d'un cours avancé sur les RdP qui s'est tenu à Bad Honnef en 1986.

La première partie de ce chapitre est consacrée à la définition des Réseaux de Petri à Objets (RPO, [Sibertin 85]), qui est le dialecte de Réseaux de Petri de Haut Niveau (RdPHN) que nous utilisons pour définir la dynamique d'une classe d'Objets Coopératifs. La deuxième partie définit plusieurs extensions originales des RPO, destinées à en accroître le pouvoir d'expression ou à rendre les modèles produits plus concis. La troisième partie constitue un rappel bibliographique, traitant des techniques de structuration des RdPHN, et des approches intégrant les concepts d'objets et de RdP.

1. LES RESEAUX DE PETRI A OBJET

Les Réseaux de Petri à Objet (RPO) [Sibertin 85] font partie de la catégorie des Réseaux de Petri de Haut Niveau, qui sont caractérisés par le fait que les jetons qui constituent le marquage des places ne sont pas des entités atomiques et indifférenciées, mais peuvent être distingués les uns des autres et portent une valeur.

La définition formelle des RPO a un pré-requis mathématique assez important, que nous rappelons dans la section II.1.1. Nous donnons ensuite, de manière très détaillée, la définition des RPO

Les RdP en général (et les RPO en particulier) bénéficient d'une représentation graphique simple et attrayante, qui facilite une compréhension intuitive du modèle ; il n'est heureusement pas nécessaire de maîtriser tous les points théoriques de la définition des RPO pour produire des modèles corrects. Le lecteur familier avec les techniques de manipulation des multiensembles et des suites finies d'éléments peut se dispenser de la lecture du préliminaire mathématique. S'il a connaissance de modèles de RdP de haut niveau tels que les Réseaux Colorés [Jensen 81, 87] ou les réseaux Prédicat-Transition [Genrich 87], il peut éviter en première lecture la définition formelle des RPO, et s'y reporter lorsque nous y faisons référence au cours du texte.

1.1. Préliminaire mathématique

a. Définition sur les ensembles

On introduit tout d'abord les notations principales qui seront utilisées tout au long des définitions. On notera :

$\mathbb{Z}$ ensemble des entiers relatifs,

$\mathbb{N}$ l'ensemble des entiers naturels,

$\mathbb{N}^*$ l'ensemble des entiers naturels non nuls,

$+$ et $\times$ les opérations d'addition et de multiplication sur les entiers,

$\leq$ la relation d'ordre "inférieur ou égal" sur les éléments de $\mathbb{Z}$,

$P(E)$ l'ensemble des parties d'un ensemble E quelconque,

$\emptyset$ l'ensemble vide.

Définition II.1 - Multiensemble

Soit E un ensemble dont les éléments sont notés e_i avec $i \in \mathbb{N}^*$

On définit un **multiensemble** sur E par une fonction x :

$x: E \rightarrow \mathbb{Z}$ de support fini (i.e. $\{i \text{ tels que } x(e_i) \neq 0\}$ est fini.)

On notera $x = \text{Erreur !}$ avec $x(e) \in \mathbb{Z}$

Un multienemble sur un ensemble E peut être considéré comme un ensemble dans lequel on a affecté un coefficient entier à chaque élément de E , déterminant la multiplicité de son appartenance.

On appelle $\mathbb{Z}(E)$ l'ensemble des multiensembles de E ($\mathbb{Z}(E)$ est le $\mathbb{Z}$ -module libre sur $\mathbb{Z}$ engendré par E),

Soient $e_0 \in E$, et $x = \text{Erreur !} \in \mathbb{Z}(E)$;

par extension de la notation ensembliste on notera :

$$e_0 \in x \text{ si et seulement si } x(e_0) \neq 0$$

$$|x| = \text{Erreur ! le cardinal du multienemble } x.$$

Un multienemble peut être aussi défini sur $\mathbb{N}$ et dans ce cas les coefficients sont positifs ou nuls. Dans les réseaux de Petri à Objets les coefficients seront uniquement pris dans $\mathbb{N}$, et on notera $\mathbb{N}(E)$ l'ensemble des multiensembles sur E à coefficients dans $\mathbb{N}$.

On peut munir $\mathbb{Z}(E)$ de deux lois de composition (l'addition notée $\oplus$ et la multiplication par un entier, notée $\otimes$) et d'une relation d'ordre partiel notée $\leq_O$.

Définition II.2 - Addition de deux multiensembles

L'addition de deux multiensembles est définie par :

$$x \oplus y = \text{Erreur !} = \text{Erreur !}$$

$\oplus$ définie ainsi est commutative, associative elle possède un élément neutre (la fonction identiquement nulle) noté $0_{\mathbb{Z}(E)}$ où tous les coefficients du multienemble sont nuls, définie par :

$$0_{\mathbb{Z}(E)}: \begin{array}{l} E \rightarrow \mathbb{Z} \\ e \rightarrow 0 \end{array}$$

Définition II.3 - Multiplication d'un multienemble par un scalaire

La multiplication d'un multienemble par un entier relatif est définie par :

$$n \otimes x = n \otimes \text{Erreur !} = \text{Erreur ! avec } n, x(e) \in \mathbb{Z}.$$

$\otimes$ définie ainsi est commutative, associative et possède un élément neutre, à savoir l'élément unité de $\mathbb{Z}$.

On peut définir $\mathbb{Z}(E)$ par récurrence à partir des deux lois $\oplus$ et $\otimes$:

$$1) \ x \in E \quad \Rightarrow \quad x \in \mathbb{Z}(E) ;$$

$$2) \ x, y \in \mathbb{Z}(E) \quad \Rightarrow \quad x \oplus y \in \mathbb{Z}(E) ;$$

$$3) \ n \in \mathbb{Z}, x \in \mathbb{Z}(E) \quad \Rightarrow \quad n \otimes x \in \mathbb{Z}(E) ;$$

Définition II.4 - Ordre partiel sur les multiensembles

La relation $\leq_O$ d'ordre partiel entre deux multiensembles est définie par :

$$\forall x, y \in \mathbb{Z}(E), x \leq_O y \quad \Leftrightarrow \quad \forall e \in E, x(e) \leq y(e)$$

Définition II.5 - Suites finies d'éléments

Soit E un ensemble. L'ensemble des **suites finies d'éléments** de E , noté E^* , se définit par :

$$E^* = \bigcup_{n \in \mathbb{N}} E^n.$$

Les éléments de E^* sont appelés n -uplets et notés $\langle e_1, \dots, e_i, \dots, e_n \rangle$, avec $n \in \mathbb{N}$, $e_i \in E$. De plus on a $E^0 = \{\langle \rangle\}$, $E^1 = E$.

Définition II.6 - Concaténation de n -uplets

La **concaténation** de n -uplets (notée $\circ$) est définie par récurrence sur E^* :

- 1) $x \in E^* \Rightarrow x \circ \langle \rangle = x$
- 2) $x \in E^*, y \in E \Rightarrow x \circ y = \langle x_1, \dots, x_n, y \rangle$

$\circ$ est associative et possède un élément neutre noté $0_{E^*} = \langle \rangle$. Ainsi défini $(E^*, \circ)$ est le monoïde libre sur E^* .

Définition II.7 - Longueur d'un n -uplet

La fonction **longueur** est définie sur E^* par récurrence :

- longueur : $E^* \rightarrow \mathbb{N}$
- 1) $\text{longueur}(0_{E^*}) = 0$;
 - 2) $x \in E^*, y \in E, \text{ longueur}(x \circ y) = \text{longueur}(x) + 1$

Définition II.8 - Multiensemble de suites finies

Soit E un ensemble. De même que dans la définition 3, on définit l'ensemble des multiensembles de suites finies de E , noté $\mathbb{Z}(E^*)$.

$\mathbb{Z}(E^*) = \{f : E^* \rightarrow \mathbb{Z} / f \text{ de support fini}\}$

$f \in \mathbb{Z}(E^*)$ si $f = \text{Erreur !}$

Exemples : Soit $E = \{x, y, z\}$ un ensemble. On a :

$$\begin{aligned} x \oplus 2 \otimes y &\in \mathbb{Z}(E); \\ \langle x, y \rangle &\in E^*; \\ \langle \langle x \rangle \oplus 2 \otimes \langle x, y \rangle \rangle &\in \mathbb{Z}(E^*) \end{aligned}$$

Soit a et $b \in \mathbb{Z}(E^*)$, avec $a = \langle \langle x \rangle \oplus 2 \otimes \langle y \rangle \rangle$ et $b = \langle \langle y \rangle \oplus 3 \otimes \langle z \rangle \rangle$. Les opérations définies ci-dessus sont utilisées ainsi :

$$\begin{aligned} a \oplus b &= (\langle \langle x \rangle \oplus 2 \otimes \langle y \rangle \rangle) \oplus (\langle \langle y \rangle \oplus 3 \otimes \langle z \rangle \rangle) = \langle \langle x \rangle \oplus 3 \otimes \langle y \rangle \oplus 3 \otimes \langle z \rangle \rangle. \\ 4 \otimes a &= 4 \otimes (\langle \langle x \rangle \oplus 2 \otimes \langle y \rangle \rangle) = (4 \otimes \langle x \rangle \oplus 8 \otimes \langle y \rangle) \end{aligned}$$

a et b ne sont pas comparables car on n'a ni $a \leq_O b$ ni $b \leq_O a$. Par contre si $a' = \{\langle x \rangle\}$ alors $a' \leq_O a$.

b. Définitions sur les applications**Définition II.9** - Prolongement canonique

Soit E et F deux ensembles et une fonction $f : E \rightarrow F$.

Soit E^* l'ensemble des n -uplets d'éléments de E .

le **prolongement canonique** f' de f est défini par :

$$\begin{aligned} f' : E^* &\rightarrow F^* \\ \langle e_1, \dots, e_n \rangle &\rightarrow f'(\langle e_1, \dots, e_n \rangle) = \langle f(e_1), \dots, f(e_n) \rangle. \end{aligned}$$

f' ainsi définie est l'unique homomorphisme de E^* vers F^* qui respecte la concaténation et prolonge f .

Définition II.10 - Prolongement linéaire

Le **prolongement linéaire** f' de f sur $\mathbb{Z}(E^*)$ est défini par.

$$\begin{aligned} f' : \mathbb{Z}(E^*) &\rightarrow \mathbb{Z}(F^*) \\ x &\rightarrow f'(x) = \text{Erreur !}, \text{ avec } x(e) \in \mathbb{Z} \end{aligned}$$

f' ainsi définie est l'unique prolongement linéaire de f qui respecte $\oplus$ et $\otimes$.

f' peut être définie par récurrence :

- 1) si $x \in E^* \Rightarrow f'(x) = f(x)$
- 2) si $x, y \in \mathbb{Z}(E^*) \Rightarrow f'(x \oplus y) = f'(x) \oplus f'(y)$ (du fait que f' est linéaire)
- 3) si $x \in \mathbb{Z}(E^*), n \in \mathbb{Z} \Rightarrow f'(n \otimes x) = n \otimes f'(x)$

Définition II.11 - Support d'un multienemble de n-uplets

La fonction **Support** associe à tout multienemble de n-uplets de E l'ensemble des éléments de E présents dans ces n-uplets. Elle peut-être définie par récurrence comme suit:

- 1) Soit 0_{E^*} élément neutre de $E^* \Rightarrow \text{support}(0_{E^*}) = \emptyset$;
- 2) Soit $x \in E \Rightarrow \text{support}(x) = \{x\}$;
- 3) Soient $x_1 \in E^*, y \in E, x = x_1 \circ y \Rightarrow \text{support}(x) = \text{support}(x_1) \cup \{y\}$;
- 4) Soient $x_1, x_2 \in \mathbb{N}(E^*), x = x_1 \oplus x_2, \text{support}(x) = \text{support}(x_1) \cup \text{support}(x_2)$;
- 5) Soient $x_1 \in \mathbb{N}(E^*), n \in \mathbb{N}^*, x = n \otimes x_1 \Rightarrow \text{support}(x) = \text{support}(x_1)$;

c. Définitions préliminaires sur les Types de Données

- On considère deux sortes de types :
 - Les types simples (INTEGER, REAL, ...)
 - Les types classe, structurés par la relation d'héritage.
- Soit C un ensemble de types; on note C_S l'ensemble des types simples de C et C_C l'ensemble des types classes de C . On a $C_S \cap C_C = \emptyset$ et $C_S \cup C_C = C$.

Définition II.12 - Domaine d'un type

On définit le **domaine** d'un type t , noté $\text{Dom}(t)$ par :

- i) si t est un type simple, $\text{Dom}(t)$ est le domaine des valeurs de t , dont les éléments sont des constantes;
- ii) si t est un type classe $\text{Dom}(t)$ est l'**ensemble des noms d'instance** de t ; Une entité de type classe sera appelée un **objet** (ou une **instance** de sa classe).

Si C est On note $U_C = \bigcup_{t \in C} \text{Dom}(t)$ l'union des domaines des valeur des types de C .

On définit également la fonction **type** :

$$\begin{array}{ccc} \text{Type} : & U_C & \rightarrow & C \\ & u & \rightarrow & t \text{ tel que } u \in \text{Dom}(t) \end{array}$$

Définition II.13 - Valeur d'une entité

A chaque élément de U_C peut être associée une **valeur** :

- i) Pour les éléments de domaine d'un type simple t ,
 $\text{Val} : \text{Dom}(t) \rightarrow \text{Dom}(t)$ est la fonction identité
- ii) Pour les éléments du domaine d'un type classe t (les noms d'objets), l'espace d'arrivée dépend de la façon dont le type d'objet est défini : La valeur d'un élément de type classe est constituée de la valeur de son nom et de la valeur de ses attributs ; ainsi deux objets ayant des attributs de même valeur mais ayant des noms différents ont une valeur différente

Définition II.14 - Compatibilité des types

On définit une relation d'ordre large partielle, sur un ensemble C de types appelée **compat**, de la manière suivante :

- i) Pour les types simples :
INTEGER compat REAL ;
CHAR compat STRING
- ii) Pour les types classe :
 C_s compat C_g si et seulement si C_s est une spécialisation de C_g suivant la hiérarchie d'héritage.

compat se généralise à C^* ainsi :

$$\langle t_1, \dots, t_n \rangle \text{ compat } \langle t'_1, \dots, t'_m \rangle \text{ si et seulement si : } n = m, \\ \text{et } \forall i \in \{1, \dots, n\}, t_i \text{ compat } t'_i$$

1.2. Définition des RPO

La définition formelle des réseaux de Petri à Objets se fonde sur le préliminaire mathématique énoncé ci-dessus.

Définition II.15 - Réseau de Petri à Objets

Un **Réseau de Petri à Objets** $R = \langle C, T, V, P, \text{Pré}, \text{Post} \rangle$ est défini par les éléments suivants :

- 1) Un ensemble C de **types**.

$U = \bigcup_{t \in T} \text{Dom}(t)$ sera appelé l'**univers** de R .

- 2) Un ensemble T de **transitions**.

- 3) Une famille $(V_t)_{t \in T}$, d'ensembles de variables, non nécessairement disjoints, indexée par T .

On note : $V = \bigcup_{t \in T} V_t$

Pour ces familles $(V_t)_{t \in T}$, on définit une famille $(\text{type}_t)_{t \in T}$, de fonctions définies par :

$$\text{type}_t : V_t \rightarrow C$$

qui s'étend (en faisant l'union avec la fonction **type** de la Définition II.12) à :

$$\text{type}_t : V_t \cup U \rightarrow C$$

$$\text{type}_t : (V_t \cup U)^* \rightarrow C^*$$

- 4) Un ensemble P de **places**, pourvu d'une fonction **type** :

$$\text{type} : P \rightarrow C^*$$

On appelle **arité** la fonction :

$$\text{arité} : P \rightarrow \mathbb{N}$$

$$p \rightarrow \text{arité}(p) = \text{longueur}(\text{type}(p))$$

De ceci on peut déduire : $\forall p \in P, \text{type}(p) \in C^{\text{arité}(p)}$.

On dira qu'une place $p \in P$ est de type **jeton simple** si $\text{arité}(p) = 0$ i.e. $\text{type}(p) = 0_{C^*}$.

Si $\text{arité}(p) > 0$, on dira que p est de type **jeton typé**.

- 5) Une fonction d'**incidence avant** Pré définie par :

$$\begin{aligned} \text{Pré} : P \times T &\rightarrow \mathbb{N}((V \cup U)^*) \\ (p, t) &\rightarrow \text{Erreur ! ; } \quad v_i \in (V \cup U)^* \end{aligned}$$

telle que :

- i) $(\text{support}(\text{Pré}(p, t)) \cap V) \subseteq V_t$;
- ii) $\forall v^* \in \text{Pré}(p, t), \text{longueur}(v^*) = \text{arité}(p)$;
- iii) $\forall v^* \in \text{Pré}(p, t), \text{type}_t(v^*) = \text{type}(p)$.

On notera $p^\bullet : \{t_i, t_i \in T, \text{Pré}(p, t_i) \neq 0\}$
 ${}^\bullet t : \{p_i, p_i \in P, \text{Pré}(p_i, t) \neq 0\}$

- 6) Une fonction d'**incidence arrière** Post définie par :

$$\begin{aligned} \text{Post} : P \times T &\rightarrow \mathbb{N}((V \cup U)^*) \\ (p, t) &\rightarrow \text{Erreur ! ; } \quad v_i \in (V \cup U)^* \end{aligned}$$

telle que :

- i) $(\text{support}(\text{Post}(p,t) \cap V) \not\subseteq V_t$;
- ii) $\forall v^* \in \text{Post}(p,t), \text{longueur}(v^*) = \text{arité}(p)$;
- iii) $\forall v^* \in \text{Post}(p,t), \text{type}_t(v^*) \text{ compat } \text{type}(p)$.

On notera $\bullet p : \{t_i, t_i \in T, \text{Post}(p,t_i) \neq 0\}$
 $t \bullet : \{p_i, p_i \in P, \text{Post}(p_i,t) \neq 0\}$

$V_{\text{in}}(t) = (\approx^p P \text{ support}(\text{Pré}(p,t)) \cap V_t)$ est l'ensemble des **variables d'entrée** d'une transition t .

$V_{\text{out}}(t) = (\approx^p P \text{ support}(\text{Post}(p,t)) \cap V_t)$ est l'ensemble des **variables de sortie** d'une transition t .

De plus on a : $V_t = V_{\text{in}}(t) \cup V_{\text{out}}(t)$

7) A chacune des transitions d'un réseau on peut associer une **précondition** :

Une précondition est une expression booléenne associée à une transition $t \in T$, dont l'ensemble des variables est inclus dans $V_{\text{in}}(t)$.

8) A chacune des transitions d'un réseau on peut associer une **action** :

Une action est un ensemble d'instructions qui sont :

soit l'**invocation**, c'est à dire l'appel d'un service publié par la classe d'un objet, noté :

`variable.opération(liste_de_paramètres)`

avec : $\text{variable} \in V_t$,

soit l'**affectation** dont la syntaxe est :

`<variable1, ..., variablen> := <expression1, ..., expressionn>` avec :

$\text{variable}_i \in V_{\text{out}}(t)$ et :

expression_i est une expression retournant une valeur du même type que variable_i et

dont les variables sont dans $V_{\text{in}}(t)$

Remarque : Dans la représentation graphique des RPO, l'arc correspondant à Pré ou Post est dessiné si et seulement si $\text{Pré}(p,t) \neq 0_{\mathbb{N}((V \cup U)^*)}$ (qui est l'élément neutre de $\mathbb{N}((V \cup U)^*)$).

a. Marquage d'un RPO

Définition II.16 - Jetons simples et jetons typés

Soit R un RPO et $U = \sum^t C \text{ Dom}(t)$ son univers.

On appelle **jeton** de R un élément $j : j \in \mathbb{N}(U^*)$.

Si $\text{longueur}(j) = 0$ on dira que j est un **jeton simple**, sinon on dira que j est un **jeton typé**.

Définition II.17 - Marquage d'un RPO

Soit R un RPO ;

Un **marquage** de R noté $M=(m, \text{Val})$ est donné par une fonction de répartition qui distribue des jetons dans les places du réseau, et par les valeurs de ces jetons.

On définit la fonction de répartition m par :

$$m : P \rightarrow \mathbb{N}(U^*)$$

$$p \rightarrow m(p) \text{ tel que :}$$

- i) $\forall p \in P, \forall j \in m(p), \text{longueur}(j) = \text{arité}(p) \text{ et } \text{type}(j) \text{ compat } \text{type}(p)$ (Le type de chaque jeton doit être compatible avec celui de la place dans lequel il se trouve).
- ii) La fonction Val est définie comme en Définition II.13

b. Franchissabilité des transitions

Définition II.18 - Substitution des variables d'une transition

Soit R un RPO, $t \in T$ une transition de R .

Une **substitution** des variables de t est une fonction S :

$$S : V_{in}(t) \rightarrow U$$

telle que : $\forall v \in V_{in}(t), \text{type}_t(v) = \text{type}(S(v))$;

Définition II.19 - Franchissabilité d'une transition

Soit (R, M) un RPO muni d'un marquage $M = (m, \text{Val})$.

Soit $t \in T$ une transition de R , et S une substitution des variables de t .

On dit que t est **franchissable** par S à partir de M (ce qui est noté $M \xrightarrow{(t, S)}$) si et seulement si :

- i) $\forall p \in P, S(\text{Pré}(p, t)) \leq m(p)$;
- ii) si $\text{Pr}(vin_1, \dots, vin_n)$ est la précondition de t alors la proposition $\text{Pr}(\text{Val}(S(vin_1)), \dots, \text{Val}(S(vin_n)))$ est vraie.

Le couple (t, S) est alors appelé une **instance de la transition** t pour le marquage M .

On dira que la transition t est **franchissable** à partir d'un marquage M (ce que l'on note $M \xrightarrow{t}$) si et seulement si $\exists S$ tel que (t, S) est une instance de t .

Sélection des Objets pour un franchissement

Les substitutions qui permettent de rendre t franchissable sont construites par un mécanisme de **semi-unification** [Robinson 79], chaque substitution associant une valeur présente dans une des places d'entrée à toute variable d'entrée de t . Si une variable est présente sur plusieurs arcs d'entrée, l'unification sur cette variable réduit les possibilités de franchissement, car la valeur substituée doit être présente dans chacune des places d'entrée.

La précondition permet ensuite d'éliminer, parmi l'ensemble des substitutions admissibles, celles qui ne vérifient pas le prédicat. Il peut rester plusieurs substitutions qui vérifient la précondition, et le choix parmi ces dernières est indéterministe.

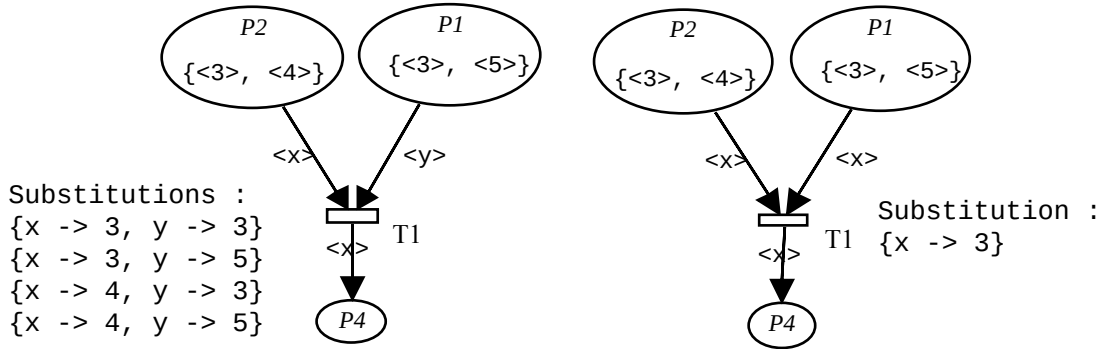


Figure II.1 - Construction des substitutions par semi-unification

La Figure II.1 montre deux transitions ne différant que par les inscriptions des arcs d'entrée. Dans le premier cas, l'ensemble des substitutions qui rendent la transition franchissable est le produit cartésien des marquages des places d'entrée, car les ensembles de variables sur les arcs d'entrée sont disjoints. Dans le second cas, l'unification sur la variable x restreint les possibilités de substitution.

Création d'objets par un franchissement

Définition II.20 - Entités créées par un franchissement

On appelle ensemble des entités créées par un franchissement l'ensemble :

$$V_{\text{new}}(t) = V_{\text{out}}(t) \setminus V_{\text{in}}(t)$$

L'occurrence de la transition peut préciser les valeurs dénotées par les variables v , telles que $v \in V_{\text{new}}(t)$

- i) Si $v \in V_{\text{new}}(t)$ est de type simple, la valeur $\text{Val}(v)$ est indéfinie à l'issue du franchissement, sauf si l'action de la transition contient une affectation $v := \text{<Expression>}$ qui fixe explicitement sa valeur.
- ii) Si $v \in V_{\text{new}}(t)$ est de type classe, on dira que la transition **instancie** la variable v . A l'issue du franchissement, v est liée un nouveau nom d'objet, qui n'existe nulle part ailleurs dans le système. La valeur de l'objet est définie par le code de l'opération de création (notée *Create*) de sa classe. L'action de la transition peut comporter des expressions de la forme $v.\text{opération}(\text{liste\_de\_paramètres})$, qui permettent de préciser l'état à l'issue du franchissement de l'instance créée.

La possibilité de générer des nouveaux noms d'objets peut être formalisée comme suit :

- Pour chaque type classe C il existe une place P_C au marquage initial infini, qui contient le nom de toutes les instances de la classe qui n'ont pas encore été instanciées ;

- Toute transition qui instancie une variable v de type classe C est reliée en entrée seulement à P_C , par un arc $\text{Pré}(P_C, t) = \langle v \rangle$ (qui n'est pas figuré sur le réseau). A l'issue du franchissement, la variable v est donc liée à un nouveau nom d'objet.

Définition II.21 - Marquage atteint par un franchissement

Soit $M \xrightarrow{(t, S)}$ une transition $t \in T$ franchissable depuis le marquage $M = (m, \text{Val})$ avec la substitution S .

$M' = (m', \text{Val}')$, le **marquage atteint** par le franchissement de la transition t avec la substitution S à partir du marquage M est défini ainsi :

- i) $\forall p \in P, m'(p) = m(p) + S(\text{Post}(p, t)) - S(\text{Pré}(p, t))$
- ii) Val' est la valeur des jetons résultant de l'application de l'action de t aux éléments de $S(V_{\text{out}}(t))$

On notera : $M \xrightarrow{(t, S)} M'$

Informellement, on peut décrire le franchissement d'une transition t de la manière suivante :

- Enlever des places de $\bullet t$ les jetons qui sont liés aux tuples de variables d'entrée ;
- Exécuter l'action de t , pour calculer les valeurs des jetons de sortie ;
- Déposer dans les places de $t \bullet$ les jetons liés aux tuples de variables de sortie.

Définition II.22 - Transitions concurremment franchissables

Soit Ψ un multiensemble d'instances de transitions.

$\Psi = \text{Erreur !}$ où (t, S) est une instance de t .

Ψ sera dit **concurremment franchissable** depuis un marquage M (noté $M \xrightarrow{\Psi}$), si et seulement si :

$$\forall p \in P, \text{Erreur !} \leq M(p)$$

Les transitions du multiensemble sont concurremment franchissables si et seulement si toutes les places en entrée des transitions contiennent suffisamment de jetons pour chacune des instances de transition de Ψ .

1.3. Analyse statique des propriétés

Le fait de permettre une analyse statique des propriétés des modèles est certainement une des caractéristiques des RdP qui a le plus contribué à développer leur utilisation. Les divers modèles de RdP de Haut Niveau, et les RPO plus particulièrement, se sont attachés à préserver ces possibilités d'analyse, tout en permettant de construire des modèles plus concis que les RdP simples.

Trois niveaux d'analyse sont possibles pour un Réseau de Petri à Objet :

- Le premier niveau traite uniquement le **réseau sous-jacent** du RPO, c'est à dire un RdP obtenu en ne gardant de la définition du RPO que ses places, ses transitions et ses arcs, et en ne conservant pour le marquage des places et les fonctions Pré et Post que le cardinal des

multiensembles qui les représentent. Le réseau sous-jacent est un RdP ordinaire, où le marquage et les fonctions d'incidences sont décrits par des entiers positifs ou nuls ;

- Le second niveau ignore les préconditions et les actions des transitions. Les classes des jetons sont prises en compte, mais leur valeur n'est pas traitée, et on ne conserve que le point i) des définitions II.17, II.19 et II.21, relatives au marquage, à la franchissabilité et au marquage atteint par un franchissement ;
- Le troisième niveau traite tous les éléments de la définition des RPO.

Pour chacun de ces niveaux, un calcul d'invariant est défini : Les invariants des RdP ordinaires sont bien connus [Lautenbach...74]. Les invariants algébriques du second niveau sont assez proches des précédents. Au troisième niveau, les invariants sont très différents, puisque ce sont des prédicats portant sur les valeurs des jetons.

La règle de franchissement étant de plus en plus contraignante du niveau 1 au niveau 3, un invariant vérifié à un niveau reste vérifié au niveau supérieur.

Le calcul des invariants des niveaux 2 et 3 est défini dans [Sibertin 85], et de manière plus détaillée dans [Sibertin 86]. Nous présentons ici, de manière succincte, les invariants algébriques de niveau 2.

Définition II.23 - Pondération d'un réseau

Selon [Jensen 82], une **pondération** (notée W) d'un RdP est un vecteur de longueur $|P|$, dont les éléments sont des fonctions

$$W = (F_p)_{p \in P}, \text{ avec :} \\ F_p : \mathbb{Z}(U^{\text{arité}}(p)) \rightarrow \mathbb{Z}(U^*)$$

où $\text{arité}(p)$ est l'arité de la place p , $\mathbb{Z}(U^*)$ est le $\mathbb{Z}$ -module généré par U^* (l'ensemble des sommes formelles d'éléments de U^* , avec des coefficients positifs ou négatifs), et F_p est une fonction linéaire dont la définition sur $\mathbb{Z}(U^{\text{arité}}(p))$ est l'extension de sa définition sur $U^{\text{arité}}(p)$.

Définition II.24 - Invariant algébrique

Une pondération W d'un réseau R est un **invariant algébrique** si et seulement si :

$\forall m \text{ et } m', \text{ deux distributions de jetons sur } R, \text{ on a : } m \rightarrow m' \Rightarrow W.m = W.m',$
avec $W.m = \text{Erreur !}$

Pour un RdP ordinaire, une pondération est un vecteur d'entiers de longueur $|P|$, et est un invariant ssi $W.C = 0$, avec $C = (\text{Post}(t, p) - \text{Pré}(t, p))_{p \in P, t \in T}$ est la matrice d'incidence du réseau. Pour les RPO (et pour les réseaux prédicat/transition également) il existe une classe de pondérations pour laquelle ce résultat est toujours valable.

Définition II.25 - Fonction uniforme

Soit E un ensemble et n un entier positif. On considère des fonctions de E^n vers E^* , qui sont des projections, des permutations ou des répétitions de composants des n -uplets :

$$F : E^n \rightarrow E^* ; \exists h \in \mathbb{N}, \exists \text{ppr} : \{1, \dots, h\} \rightarrow \{1, \dots, n\} \\ \forall \langle e_1, \dots, e_n \rangle \in E^n, F(\langle e_1, \dots, e_n \rangle) = \langle e_{\text{ppr}(1)}, \dots, e_{\text{ppr}(h)} \rangle.$$

Une fonction $F : E^n \rightarrow \mathbb{Z} (E^*)$ est **uniforme** si elle est une combinaison linéaire de telles fonctions.

Proposition

Soit $W = (F_p)_{p \in P}$ une pondération pour laquelle les F_p sont uniformes.

W est un invariant algébrique de ce réseau si et seulement si $W.C = 0$.

Nous n'avons pas connaissance d'un algorithme qui calcule tous les invariants algébriques d'un réseau. Toutefois, si des fonctions uniformes F_p sont fournies, les techniques de calcul des invariants des RdP peuvent être utilisées pour calculer les vecteurs d'entiers $(i_p)_{p \in P}$ tels que $W.C = 0$, avec $W = (i_p.F_p)_{p \in P}$.

Ces invariants sont moins puissants que les S-invariants de [Genrich...86], mais sont plus faciles à calculer.

2. EXTENSION DES RPO

Nous allons dans ce chapitre introduire un certain nombre d'extensions à la définition II.15 des Réseaux de Petri à Objets. Certaines de ces extensions sont nouvelles (places triées, arcs inhibiteurs généralisés), d'autres ont déjà été introduites dans différents dialectes de RdP. Ces dernières n'ont pas toutes reçu une définition qui prenne en compte les nouvelles possibilités offertes par les RdP de Haut Niveau, et plus particulièrement par les RPO, aussi en proposerons nous de nouvelles définitions.

Ces extensions peuvent être classées en trois catégories, en fonction de leur objectif :

- Certaines (*registres*, *règles d'émission* et *places à capacité*) sont uniquement des **abréviations**, dont le but est d'accroître la concision des modèles en fournissant en tant que primitives des constructions fréquemment utilisées. Pour chacune de ces extensions, il existe un algorithme très simple permettant de se ramener à un réseau équivalent, mais qui utilise uniquement la définition de base des RPO. Les abréviations n'augmentent donc pas le **pouvoir d'expression** du formalisme. La concision et la lisibilité étant des facteurs clef de la valeur d'usage d'un formalisme, ces abréviations sont largement utilisées dans le reste de ce mémoire ;
- D'autres extensions (*arcs à priorité*, *arcs inhibiteurs*, *ordonnancement des places*) ont pour objectif de donner au concepteur des moyens de maîtriser le **non déterminisme** des modèles ; un réseau de Petri simple peut décrire des situations de choix non-déterministe : c'est le cas chaque fois que deux transitions en **conflit structurel** sont en **conflit effectif** pour un marquage donné ;

Les extensions qui visent à permettre une maîtrise de l'indéterminisme ne sont pas de simples abréviations (elles ne sont pas toujours réductibles, en dehors de cas simples, à un RPO normal) et augmentent effectivement le pouvoir d'expression du formalisme, au détriment des possibilités d'analyse statique des modèles. Toutefois, leur introduction nous paraît justifiée, dans l'optique d'une utilisation du formalisme pour la modélisation de systèmes en grandeur réelle. Elles ont cependant été utilisées avec parcimonie dans ce mémoire ;

- Enfin l'objectif de la dernière extension (la *temporisation*) est explicitement d'accroître le pouvoir d'expression du formalisme, afin de le rendre applicable à une classe plus large de problèmes.

Ces extensions ne sont pas toutes orthogonales entre elles (les priorités, par exemple, sont exprimables par des arcs inhibiteurs et réciproquement). Toutefois les cas d'utilisation pratique de telles extensions sont suffisamment différents pour justifier de cette redondance.

2.1. Indéterminisme dans les Réseaux de Petri à Objets

Plusieurs de ces extensions ont pour objectif de réduire l'indéterminisme de l'évolution d'un réseau, aussi nous commençons par caractériser les différents niveaux d'indéterminisme présents dans un RPO.

Définition II.26 - Conflit structurel et conflit effectif

Deux transitions t_1 et t_2 sont en **conflit structurel** [Brams 83] si et seulement si elles ont au moins une place commune en entrée :

$$\exists p : \text{Pré}(p, t_1) \neq 0 \text{ et } \text{Pré}(p, t_2) \neq 0.$$

Elles sont en **conflit effectif** pour un marquage M si de plus :

$$M \not\vdash t_1 \rightarrow, M \not\vdash t_2 \rightarrow \text{ et } \exists p : M(p) < \text{Pré}(p, t_1) + \text{Pré}(p, t_2)$$

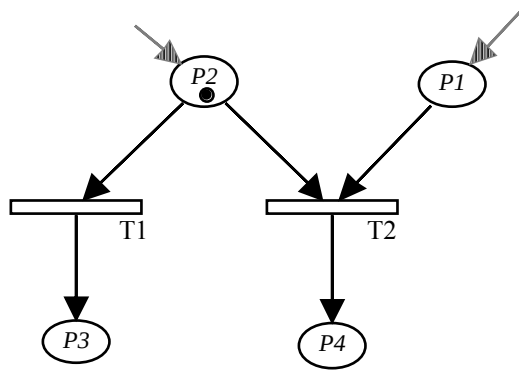


Figure II.2 - conflit de transitions

En cas de conflit effectif entre deux transitions, et en l'absence de toute interprétation, le choix entre ces transitions est non déterministe. A titre d'exemple, dans la Figure II.2 les transitions $T1$ et $T2$ sont en conflit structurel ; avec le marquage donné sur la figure, les deux transitions ne sont pas en conflit effectif, car seule $T1$ est franchissable. Le conflit sera effectif pour tout marquage M tel que : $M(P1) \geq 1$ et $M(P2) = 1$

Lorsque l'on considère des réseaux de haut niveau, où les jetons sont différenciables (réseaux à individus) ou portent des valeurs (réseaux à prédicats ou à objets), un niveau supplémentaire d'indéterminisme vient s'ajouter : il peut exister plusieurs manières de franchir une transition donnée, avec des substitutions capturant des individus différents dans les places d'entrée.

La définition des RPO contient un mécanisme permettant de réduire cet indéterminisme : il s'agit des préconditions, associées aux transitions du réseau (§II.1.2). Toutefois les préconditions peuvent s'avérer insuffisantes dans de nombreuses situations, puisqu'elles peuvent uniquement tester ou apparier des valeurs déjà substituées aux variables d'entrée de la transition, et non pas faire des choix portant sur la globalité du marquage d'une place, ou des choix non locaux à une seule transition.

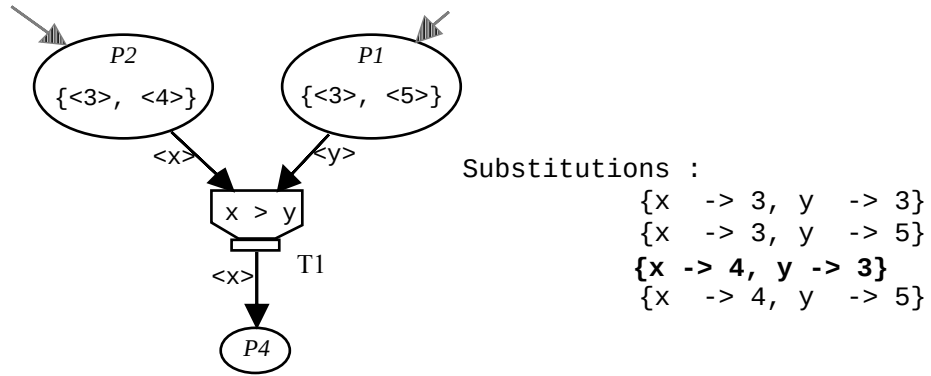


Figure II.3 - Indéterminisme lié aux valeurs des jetons

La Figure II.3 illustre l'indéterminisme lié aux valeurs des jetons : sans précondition, la transition $T1$ est franchissable avec les quatre substitutions citées en légende. L'indéterminisme est levé par la précondition, qui autorise le franchissement uniquement avec la substitution $\{x \rightarrow 4, y \rightarrow 3\}$

2.2. Registres

Lorsque l'on décrit un système, il est fréquent de rencontrer des variables d'état dont la valeur est à la fois souvent nécessaire et toujours disponible. Si le système évolue de manière concurrente, l'accès et la modification sur ces variables doivent être effectués en exclusion mutuelle. Une telle variable est souvent désignée comme une **variable globale** d'un système. Pour modéliser de telles variables, on inclut dans la définition d'un RPO la donnée d'un ensemble de registres.

Définition II.27 - Registres d'un RPO

Soit $p \in P$. p est une **place registre** si et seulement si :

$\forall t \in T, t \in p^\bullet \Leftrightarrow t \in {}^\bullet p$ (Toutes les transitions en entrée de p figurent également en sortie)

$\forall t_i, t_j \in p^\bullet, \text{Pré}(p, t_i) = \text{Pré}(p, t_j) = \text{Post}(p, t_i) = \text{Post}(p, t_j)$ (Les étiquettes des arcs adjacent à p sont toutes identiques)

p peut être omise sur la représentation graphique du réseau, ainsi que tous les arcs qui lui sont adjacents (Figure II.4). On introduit à sa place dans la définition du réseau un **registre** :

$\text{Reg} = (\text{Nom}, \text{Type}, \text{Val\_init})$ avec :

$\text{Nom} = \text{Pré}(p, t_i)$,

$\text{Type} = \text{Type}(p)$,

$\text{Val\_init} = m_0(p)$

Cette construction a pour inconvénient de dissimuler le fait que deux transitions utilisant le jeton de la place registre ne peuvent pas être franchies concurrentement. Ceci nous paraît largement compensé par l'allègement de la représentation graphique qui en résulte, et qui est illustré en Figure II.4.

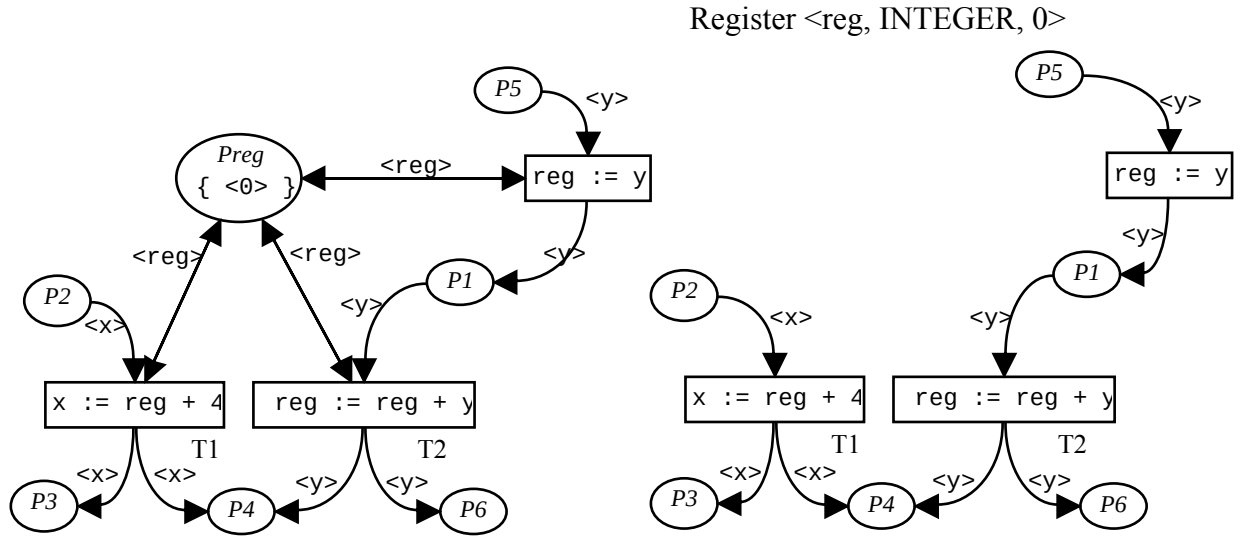


Figure II.4 - Place registre et notation abrégée

2.3. Règles d'émission

Le formalisme utilisé dans la méthode Merise [Tardieu...83] pour décrire la dynamique des processus utilise le mécanisme de **règles d'émission** pour exprimer le fait qu'une opération ne se termine pas toujours de la même manière, mais peut fournir des résultats différents en fonction des valeurs avec lesquelles elle est franchie.

Définition II.28 - Règle d'émission

On, appelle **règle d'émission** une place $p \in P$, telle que :

$|\bullet p| = 1$ (La place n'a qu'une transition d'entrée)

$\forall t \in p^\bullet, t = \{p\}$

$\Delta_{t \in p^\bullet} \text{Précond}(t) = \text{TRUE}$

(la disjonction des préconditions des transitions en sortie de p est une tautologie)

Pour alléger la représentation des RPO, on associe aux règles d'émission un motif graphique qui masque la place, et regroupe sa transition d'entrée et ses transitions de sortie en une macro-transition (Figure II.5). Pour assurer que la disjonction des préconditions soit une tautologie, on peut mentionner le mot clef *ELSE*, qui correspond alors à la négation de la disjonction des autres prédicats.

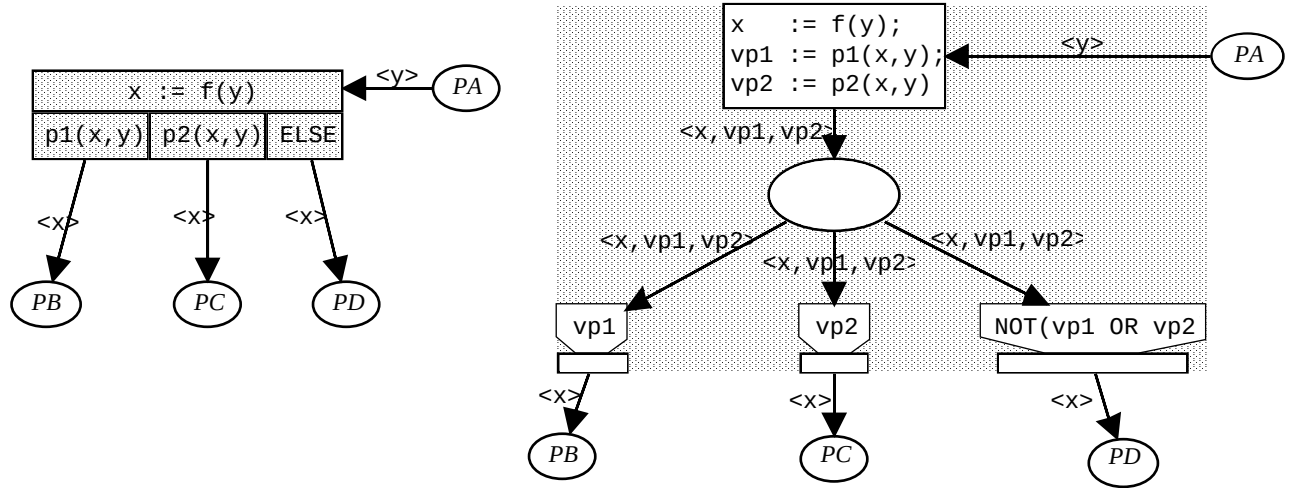


Figure II.5 - Règle d'émission et notation abrégée

Il faut remarquer que les préconditions des transitions de sortie du motifs sont évaluées à l'occurrence de la transition d'entrée, ce qui assure que la franchissabilité des transitions de sortie est indépendante des franchissements qui ont pu avoir lieu dans le réseau depuis l'occurrence de la transition d'entrée.

2.4. Places à capacité

Lorsqu'on modélise un système par RdP, il est fréquent d'interpréter une ou plusieurs places du réseau comme des éléments de stockage physique. La capacité d'un tel élément (i.e. le nombre d'objets qu'il peut contenir) est nécessairement limitée. Il est donc commode d'associer à la place représentant cet élément une **capacité** $n \in \mathbb{N}$ ($n > 0$) spécifiant le nombre de jeton maximum qu'elle peut contenir. La définition de la franchissabilité est alors modifiée de la manière suivante :

$t \in T$ est franchissable depuis M par la substitution S si :

- 1) $\forall p \in P, S(\text{Pré}(p,t)) \leq m(p)$
- 2) $|m(p)| \leq \text{Capacité}(p) - |\text{Post}(p,t)| + |\text{Pré}(p,t)|$

spécifiant ainsi qu'aucun franchissement ne peut rendre le cardinal du marquage d'une place supérieur à sa capacité.

Le procédé permettant de transformer un réseau à capacité en un réseau simple est décrit dans [André 81, Jantzen...79]. Il consiste simplement à rajouter, pour chaque place p à capacité, une place p' de type jeton, dite **place complémentaire** de p , dont les transitions d'entrée et de sortie sont respectivement les transitions de sortie et d'entrée de la place p .

Le marquage initial de la place complémentaire est défini par :

$$m_0(p') = (\text{Capacité}(p) - |m_0(p)|) \cdot \diamond$$

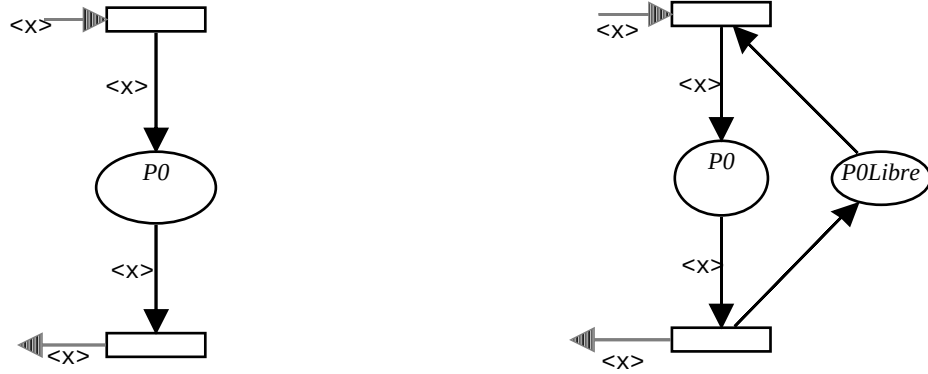


Figure II.6 - Place à capacité et notation abrégée

Graphiquement, on représentera la capacité d'une place entre parenthèses, à l'intérieur de l'ellipse qui la représente. La Figure II.6 illustre la syntaxe graphique d'une place à capacité, et le RPO équivalent.

2.5. Arcs à priorité

Les réseaux à priorité [Hack 75] ont pour objectif de réduire l'indéterminisme dans l'évolution d'un réseau en munissant l'ensemble T des transitions d'une relation d'ordre partiel $<$. La franchissabilité des transitions est alors définie de la manière suivante :

Définition II.29 - Franchissabilité d'un RdP à priorités

$t \in T$ est franchissable depuis M par la substitution S si :

- 1) $\forall p \in P, S(\text{Pré}(p,t)) \leq m(p)$
- 2) il n'existe pas $t' \in T$ tel que : $M \xrightarrow{t'} \text{ et } t < t'$

Toutefois, il est en général peu significatif d'affecter une priorité globalement à toutes les transitions du réseau, et il n'existe pas de représentation graphique lisible des priorités sur les transitions. En pratique, les priorités sont surtout utilisées pour lever l'indéterminisme entre deux transitions en conflit structurel, et cette utilisation nous apparaît également comme étant la mieux fondée méthodologiquement : il semble en effet que le fait de gérer des priorités sur l'ensemble des transitions du réseau contrevienne à l'aspect strictement local de la règle de franchissement, où seules les places adjacentes à la transition en cours d'examen sont à considérer.

Pour favoriser cette utilisation des priorités, nous proposons donc de réduire la portée de la définition de Hack, en affectant des priorités non plus aux transitions, mais aux **arcs** du réseau, et en se limitant à deux niveaux de priorité : on parlera alors d'**arcs prioritaires**, par opposition aux **arcs sans priorité**.

La relation d'ordre est alors calculée en fonction de la priorité des arcs.

Définition II.30 - RPO à priorités

Un **RPO à priorités** $R = \langle C, T, V, P, \text{Pré}, \text{Post}, \text{Prio} \rangle$ est défini par l'adjonction à un RPO $\langle C, T, V, P, \text{Pré}, \text{Post} \rangle$ d'une fonction Prio :

$$\text{Prio} : \text{PxT} \rightarrow \{0, 1\}$$

telle que la valeur de $\text{Prio}(p,t)$ n'est significative que si $\text{Pré}(p,t) \neq 0$; graphiquement, un arc $\text{Pré}(p,t)$ tel que $\text{Prio}(p,t) = 1$ sera représenté par un trait en gras.

La priorité d'une transition t est alors définie par :

$$\text{Prio}(t) = \text{Erreur !}$$

On se ramène alors à la définition II.29 de Hack par :

$$\forall t, t' \in T, \quad t < t' \Leftrightarrow \text{Prio}(t) < \text{Prio}(t')$$

La Figure II.7 (partie gauche) illustre la définition explicite des priorités sur les transitions, et la définition implicite par l'utilisation des arcs prioritaires (partie droite). La représentation de droite est plus lisible, les arcs en gras étant intuitivement interprétés comme le "chemin privilégié" que les jetons cherchent à parcourir. Dans les deux cas, si $T1$ et $T2$ sont simultanément franchissables, c'est $T1$ qui sera choisie.

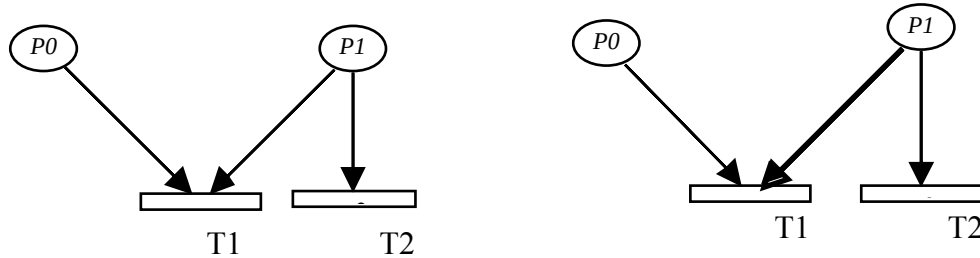


Figure II.7 - Priorités par transitions et priorités par arcs

Notre définition est par ailleurs équivalente à celle de Hack : en effet, quelle que soit la relation d'ordre sur les transitions pour un réseau donné, il est toujours possible d'émuler cette relation d'ordre par notre notation en rajoutant des places à cet effet.

Le mécanisme de priorité sur les arcs se combine élégamment avec les places à capacité, au prix de l'introduction d'une nouvelle notation graphique : l'arc en gras figure alors en **sortie** d'une transition, mais la sémantique d'une telle notation illustrée en Figure II.8, est d'appliquer la priorité sur l'arc entrant adjacent à la place complémentaire. On rencontrera de telles constructions dans l'étude de cas du §VII.1.

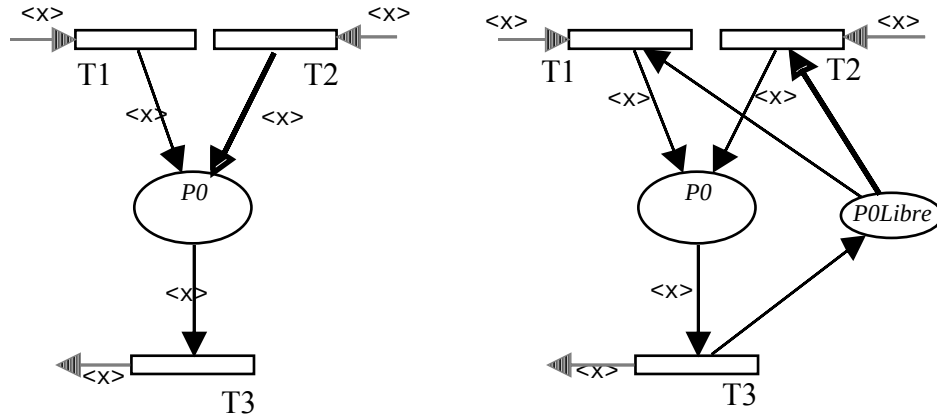


Figure II.8 - Priorités sur les arcs et place à capacité

2.6. Arcs inhibiteurs généralisés

Les arcs inhibiteurs [Hack 75] ont été introduits afin de permettre le **test à zéro** dans un RdP, c'est à dire conditionner la franchissabilité d'une transition au fait qu'une place du réseau est vide. Il a été démontré [Brams 83, Peterson 81] que cette extension dote les RdP de la puissance d'expression d'une machine de Turing, et rend donc leur finitude indécidable.

Dans le cadre de RPO, nous allons étendre la sémantique des arcs inhibiteurs classiques en introduisant les **arcs inhibiteurs généralisés**, qui permettent d'exprimer que le franchissement d'une transition est conditionné non plus par le fait qu'une place est vide, mais par le fait qu'une place ne contient pas de jeton portant une certaine valeur. Si l'on considère que les arcs inhibiteurs classiques permettent d'exprimer la négation en logique des propositions, les arcs inhibiteurs valués vont permettre d'exprimer la négation en logique des prédicats.

Nous donnons tout d'abord une définition des arcs inhibiteurs au niveau des RdP simples. Cette définition est une alternative à celle de Hack, mais est légèrement plus expressive. Nous étendons ensuite cette définition aux RPO.

Rappelons tout d'abord la définition originale :

Définition II.31 - Arcs inhibiteurs ([Hack 75])

Un RdP à **arcs inhibiteurs** est un RdP $R = \langle P, T, \text{Pré}, \text{Post} \rangle$, dont on modifie seulement la définition de Pré :

$$\text{Pré} : P \times T \rightarrow \mathbb{N} \cup \{ \emptyset \}$$

Dans un RdP à arcs inhibiteurs, une transition $t \in T$ est franchissable pour un marquage M si :

$$\forall p \in P : \text{Pré}(p, t) \leq M(p) \quad \text{si } \text{Pré}(p, t) \in \mathbb{N}$$

$$M(p) = 0$$

$$\text{si } \text{Pré}(p, t) = \emptyset$$

Les éléments tels que $\text{Pré}(p, t) = \emptyset$ sont représentés graphiquement par un arc où un petit rond remplace la flèche. La Figure II.9 illustre cette représentation : la transition T2 ne peut être franchie que si la place P1 ne contient pas de jeton. La légende montre la représentation intuitive que l'on peut

avoir, en logique propositionnelle, d'une telle construction ; dans ce cas, la proposition P_x est vraie si $M(P_x) > 0$.

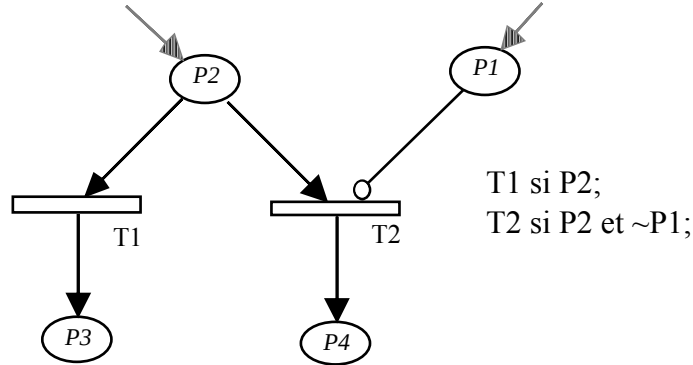


Figure II.9 - Arc inhibiteur

Nous proposons une autre définition des arcs inhibiteurs :

Définition II.32 - Arcs inhibiteurs

Un RdP à **arcs inhibiteurs** $R = \langle P, T, \text{Pré}, \text{Post}, \text{Inhib} \rangle$ est défini par l'adjonction à un RdP $\langle P, T, \text{Pré}, \text{Post} \rangle$ d'une fonction Inhib :

$\text{Inhib} : P \times T \rightarrow \mathbb{N} \cup \{ \omega \}$ (complétion de $\mathbb{N}$ par un plus grand élément ω)

Dans un RdP à arcs inhibiteurs, une transition $t \in T$ est franchissable pour un marquage M si et seulement si :

- 1) $\forall p \in P : \text{Pré}(p, t) \leq M(p)$
- 2) $\forall p \in P : \text{Inhib}(p, t) > M(p)$

La définition du marquage M' atteint depuis M par occurrence de t est identique à celle des RdP simples. Si $\text{Inhib}(p, t) = \omega$, p n'impose aucune contrainte sur le franchissement de t , de même que lorsque $\text{Pré}(p, t) = 0$. Un arc (p, t) tel que $\text{Inhib}(p, t) = 1$ fonctionne comme un arc inhibiteur de Hack.

Cette définition est plus puissante que celle de Hack, puisqu'elle autorise les arcs inhibiteurs à être pondérés, permettant alors de définir qu'une transition est franchissable quand une place contient moins d'un certain nombre de jetons. Elle permet en outre, comme illustré en Figure II.10, de combiner arc inhibiteur et arc standard entre une place et une transition, spécifiant dans cet exemple que la transition $T1$ n'est franchissable que si la place $P1$ contient un jeton et un seul.

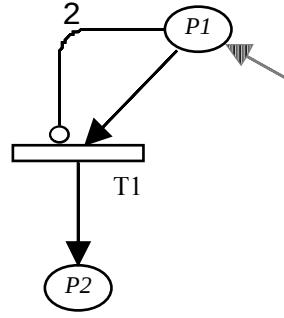


Figure II.10 - combinaison d'arc inhibiteur et d'arc standard

L'extension de la Définition II.32 aux RPO est très simple, la fonction *Inhib* subissant le même enrichissement que la fonction *Pré* (§II.1.2).

Définition II.33 - RPO à arcs inhibiteurs

Un **RPO à arcs inhibiteurs** $R = \langle C, T, V, P, \text{Pré}, \text{Post}, \text{Inhib} \rangle$ est défini par l'adjonction à un RPO $\langle C, T, V, P, \text{Pré}, \text{Post} \rangle$ d'une fonction *Inhib* :

$$\text{Inhib} : P \times T \rightarrow \mathbb{N}((V \cup U)^*)$$

telle que :

- i) $(\text{support}(\text{Inhib}(p, t)) \cap V) \not\subseteq V_t$
- ii) $\forall v^* \in \text{Inhib}(p, t), \text{longueur}(v^*) = \text{arité}(p);$
- iii) $\forall v^* \in \text{Inhib}(p, t), \text{type}_t(v^*) = \text{type}(p).$

L'ensemble des variables d'entrée d'une transition est alors défini comme :

$$V_{\text{in}}(t) = \approx, p \text{ }^P \text{ support}(\text{Pré}(p, t)) \cup \approx, p \text{ }^P \text{ support}(\text{Inhib}(p, t))$$

Définition II.34 - Franchissabilité dans un RPO à arcs inhibiteurs

Dans un RPO à arcs inhibiteurs, une transition $t \in T$ est **franchissable** à partir d'un marquage $M(m, \text{Val})$ si et seulement si :

il existe une substitution $S \in \mathbb{S}$ pour t

$$s : V_{\text{in}}(t) \rightarrow U \text{ telle que :}$$

- 1) $\forall p \in P, S(\text{Pré}(p, t)) \leq m(p);$
- 2) si $\text{Pr}(v_1, \dots, v_n)$ est la précondition de t alors :
 $\text{Pr}(\text{Val}(S(v_1)), \dots, \text{Val}(S(v_n))) \Rightarrow \forall p \in P, S(\text{Inhib}(p, t)) \leq, / m(p)$ ³.

Il faut remarquer que si la transition n'a pas de précondition (ou plus précisément si sa précondition est identiquement vraie), le point 2) de la définition précédente se simplifie en :

$$\forall p \in P, S(\text{Inhib}(p, t)) \leq, / m(p).$$

³ On note $\leq, /$ et non pas $>$, car l'ordre n'étant pas total deux éléments peuvent être non comparables.

Pour qu'un arc inhibiteur n'impose aucune contrainte au franchissement, il doit être de la forme : **Erreur !**, où v_i n'apparaît pas sur les autres arcs (v_i peut alors être instancié par toute substitution).

Le cas d'un arc inhibiteur sans précondition est illustré en Figure II.11 : en l'absence de l'arc inhibiteur, la transition $T2$ est franchissable avec les substitutions $\{a \rightarrow 3\}$, $\{a \rightarrow 4\}$ et $\{a \rightarrow 5\}$; l'arc inhibiteur réduit les possibilités de franchissement à la substitution $\{a \rightarrow 5\}$

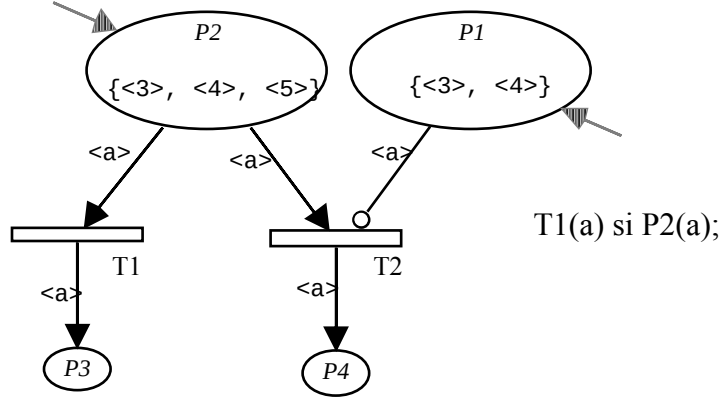


Figure II.11 - Arc inhibiteur généralisé sans précondition

La Figure II.12 montre la combinaison d'un arc inhibiteur avec une précondition. Si on donne, par exemple, $\text{Prop}(a,b) \Leftrightarrow ((a + b) < 7)$, la transition $T2$ est franchissable avec les substitutions $\{b \rightarrow 4\}$ et $\{b \rightarrow 5\}$. Elle n'est pas franchissable avec la substitution $\{b \rightarrow 3\}$, car il existe dans $P1$ une valeur de a ($\{a \rightarrow 3\}$) telle que $\text{Prop}(a, b)$ est vérifiée.

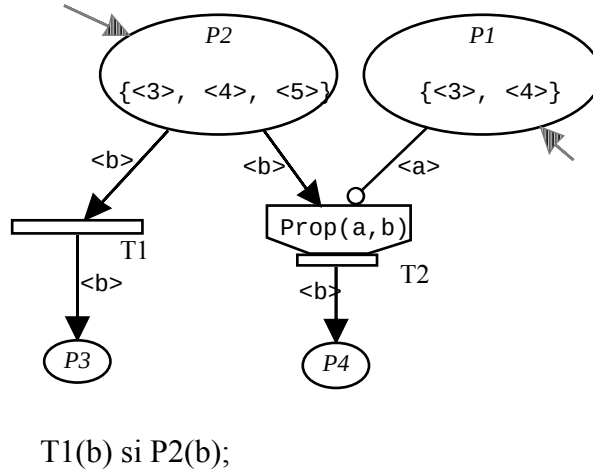
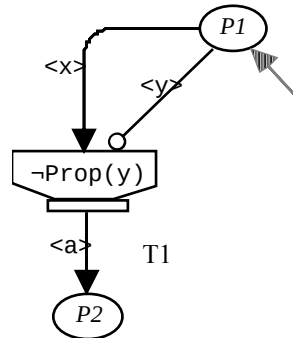


Figure II.12 - Arc inhibiteur généralisé avec précondition

Dans ces figures, on a fait figurer en légende des notations logiques visant à faciliter une compréhension intuitive de la franchissabilité des transitions. Une formule du type $Tx(a) \text{ si } Py(a)$ doit être lue "Quel que soit a , la transition Tx est franchissable avec la valeur a si le marquage de la place Py contient le jeton $\langle a \rangle$ ".

Il faut remarquer que l'introduction d'un arc inhibiteur inverse la signification habituelle des préconditions : ainsi une transition avec arc inhibiteur et sans précondition est plus contrainte que la même transition avec une précondition.



$T1(x)$ si $P1(x)$ et $\neg(P1(y) \text{ et } \neg\text{Prop}(y))$;

$T1(x)$ si $P1(x)$ et $\text{Prop}(y)$;

Figure II.13 - Arc inhibiteur testant tous les jetons d'une place

La Figure II.13 illustre la puissance d'expression de cette extension : la transition $T1$ n'est franchissable que si **tous les jetons** dans $P1$ vérifient le prédicat Prop . Nous réutiliserons cette technique dans la section suivante, en montrant comment ordonner les jetons à l'intérieur des places d'un réseau.

2.7. Ordonnancement des jetons du marquage

Lorsque le marquage des places est constitué d'entités qui ont une valeur et non pas de jetons indifférenciés, il peut être pertinent de se préoccuper de l'ordre dans lequel les jetons vont quitter la place par occurrence d'une transition. Cet ordre peut être défini en fonction de la valeur des jetons (on parlera alors de **place triée**), ou en fonction de leur ordre d'arrivée dans la place (place gérée comme une pile ou comme une file).

Définition II.35 - Critère de tri d'une place

On appelle **critère de tri** d'une place p une fonction Tri_p :

$$\text{Tri}_p : \text{Type}(p) \rightarrow D$$

où D est un domaine sur lequel est définie une relation d'ordre (par exemple les entiers, les chaînes de caractères, ...). La fonction Tri_p permet d'ordonner les jetons présents dans la place suivant un critère dépendant de leur valeur.

Définition II.36 - Place triée

Une place p sera dite **non triée** si Tri_p est une fonction constante, ou si aucun critère de tri ne peut lui être associé. Dans le cas contraire, p sera dite **place triée**.

On ne peut associer un critère de tri Tri_p à une place p que si :

$$\forall t \in T, \text{Pré}(p,t) \in (V \cup U)^*$$

(c'est à dire que tout arcs sortant de p est étiqueté par un seul tuple, et non par un multienemble de tuples).

Le tri d'une place p sert à sélectionner les jetons susceptibles d'être utilisés pour le franchissement d'une transition t en sortie de p . On peut envisager deux interprétations pour le tri :

- Tri **fort** (hard) : Le tri à lieu avant l'étape de filtrage, et peut bloquer la franchissabilité de t : On ne peut construire une substitution qu'avec le jeton qui optimise le critère de tri;
- Tri **faible** (soft) : Le tri sert à choisir une substitution parmi celles qui rendent t franchissable : on franchit avec la substitution qui optimise le critère de tri.

On verra que ces deux interprétations ne sont équivalentes que si la transition t n'a pas de précondition.

Dans les deux cas, le tri peut être **ascendant** (Ascending) si on cherche à minimiser le critère de tri, ou **descendant** (Descending) dans le cas contraire.

Une place p , fortement triée suivant la fonction Tri_p , sera notée :

$$p : \langle v_1 : C_1, \dots, v_n : C_n \rangle \text{ **Hard Ascending** } \text{Tri}_p \langle v_1, \dots, v_n \rangle \text{ ou :}$$

$$p : \langle v_1 : C_1, \dots, v_n : C_n \rangle \text{ **Hard Descending** } \text{Tri}_p \langle v_1, \dots, v_n \rangle$$

Un place p , faiblement triée suivant la fonction Tri_p , sera notée :

$$p : \langle v_1 : C_1, \dots, v_n : C_n \rangle \text{ **Soft Ascending** } \text{Tri}_p \langle v_1, \dots, v_n \rangle \text{ ou :}$$

$$p : \langle v_1 : C_1, \dots, v_n : C_n \rangle \text{ **Soft Descending** } \text{Tri}_p \langle v_1, \dots, v_n \rangle$$

Avant de caractériser formellement l'influence du type de tri sur la franchissabilité des transitions, nous illustrons les tris forts et faibles à l'aide de deux exemples :

Exemples :

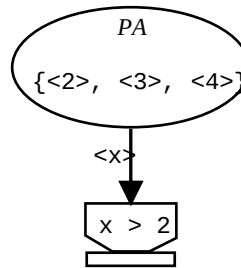


Figure II.14 - Influence du tri sur la franchissabilité

Dans l'exemple de la Figure II.14, si la place PA n'est pas triée, ou est faiblement triée, la transition T1 est franchissable avec les substitutions $\{x \rightarrow 3\}$ et $\{x \rightarrow 4\}$; par contre, si la place est fortement triée (Hard Ascending x), la transition n'est pas franchissable, car la substitution qui minimise le critère de tri ($\{x \rightarrow 2\}$) ne vérifie pas la précondition.

Dans la Figure II.15, la transition T1 est franchissable avec les substitutions $\{x \rightarrow 3\}$ et $\{x \rightarrow 4\}$ si aucune place n'est triée; elle est franchissable uniquement avec la substitution $\{x \rightarrow 3\}$ si la place PA est déclarée Soft Ascending; si la place est déclarée Hard Ascending, la transition n'est pas franchissable, car le jeton $\langle 2 \rangle$, qui minimise le critère de tri, n'est pas présent dans la place PB.

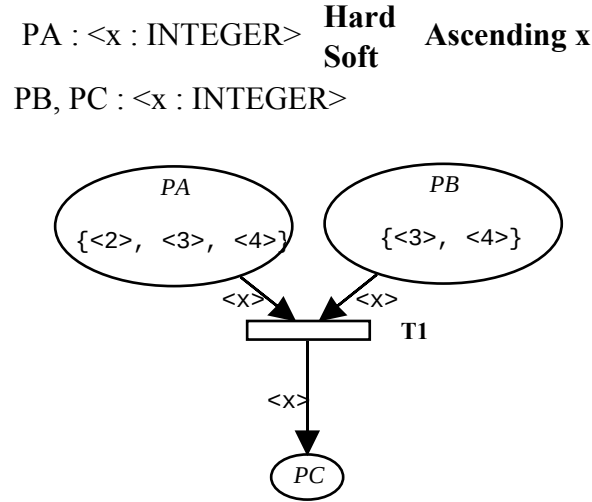


Figure II.15 - Tri et unification

Définition II.37 - Places triées et franchissabilité

Soit (R, M) un RPO muni d'un marquage $M(m, \text{Val})$.

Soit $t \in T$ une transition de R , et S une substitution des variables de t .

On dit que t est **franchissable** par S à partir de M (ce qui est noté $M \xrightarrow{(t, S)}$) si et seulement si :

- 1) $\forall p \in P, S(\text{Pré}(p, t)) \leq m(p)$;
- 2) si $\text{Pr}(\text{vin}_1, \dots, \text{vin}_n)$ est la précondition de t alors la proposition $\text{Pr}(\text{Val}(S(\text{vin}_1)), \dots, \text{Val}(S(\text{vin}_n)))$ est vraie.
- 3) $\forall p \in P$, telle que $\text{Pré}(p, t) \neq 0$ et Tri_p est un critère de tri **fort ascendant** (resp. **descendant**), $\forall j \in m(p)$:

$$\text{Tri}_p(\text{Val}(S(\text{Pré}(p, t)))) \leq \text{Tri}_p(j)$$
(resp. $\text{Tri}_p(\text{Val}(S(\text{Pré}(p, t)))) \geq \text{Tri}_p(j)$)
- 4) $\forall p \in P$, telle que $\text{Pré}(p, t) \neq 0$ et Tri_p est un critère de tri **faible ascendant** (resp. **descendant**), $\forall S'$ tel que $S'(\text{Pré}(p, t)) \leq m(p)$,

$$\text{Tri}_p(\text{Val}(S(\text{Pré}(p, t)))) \leq \text{Tri}_p(\text{Val}(S'(\text{Pré}(p, t))))$$
(resp. $\text{Tri}_p(\text{Val}(S(\text{Pré}(p, t)))) \geq \text{Tri}_p(\text{Val}(S'(\text{Pré}(p, t))))$)

Pour une transition t donnée, il n'y a pas de différence entre le tri faible et fort de ses places d'entrée si t ne comporte pas de précondition, et s'il n'y a pas de contrainte d'unification sur ses variables d'entrée

(i.e. si chaque variable d'entrée ne figure que sur un seul arc). Dans ce cas, en effet, les jetons qui optimisent un critère de tri se trouvent forcément dans une des substitutions possibles.

Nous allons montrer que les places fortement triées (mention Hard) sont une simple abréviation des RPO, et nous allons décrire des motifs permettant de les émuler. Les places faiblement triées (mention Soft), par contre, sont une extension du modèle, puisque le choix se fait alors parmi l'ensemble des substitutions possibles. Leur introduction étend donc le pouvoir d'expression du formalisme.

Nous commençons par la présentation de deux abréviations des RPO permettant d'exprimer qu'une place est gérée en pile ou en file [Sibertin 87].

a. Place file

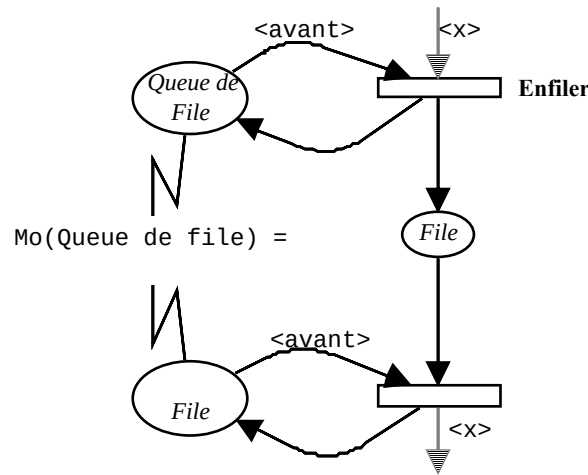


Figure II.16 - Un motif se comportant comme une file

Une place peut modéliser une file d'attente, une des structures de base de la technique informatique, exprimant le fait que les jetons seront capturés lors du franchissement des transitions de sortie dans l'ordre où ils ont été déposés dans la place par les transitions d'entrée. On connaît ce mode de gestion sous son appellation anglaise de FIFO (First In, First Out).

Le fait qu'une place doit être gérée en FIFO sera précisé dans sa définition textuelle par la mention **Hard FIFO**. Une telle déclaration donne lieu à l'extension illustrée par la Figure II.16.

Cette abréviation fait un grand usage de l'unification et de la génération de noms d'objets. C'est un bon exemple de la puissance d'expression des RPO, qui mérite une explication détaillée :

Les valeurs des objets présents dans les places *Tête_de_file* et *Queue_de_File* ne sont d'aucune utilité : seule l'**identité** de ces objets est importante, puisque elle sert à l'unification sur la transition *Défiler*. Ces deux places sont donc définies de type $\langle x : \text{ANY} \rangle$, ou $\text{ANY} \in C$ est une classe d'objets quelconque (on pourrait choisir la racine de la hiérarchie d'héritage). Une autre solution, moins élégante mais qui utilise des mécanismes plus simples, serait de stocker dans ces places des entiers, la transition *enfiler* exécutant l'action après $:= \text{avant} + 1$.

La transition *Enfiler* a une variable de sortie (*après*) qui n'est pas une variable d'entrée : d'après la définition II.20 ceci correspond à la création d'une nouvelle instance de la classe *ANY*, dont le nom est déposé dans la place *Tête de File*.

Pour que ce motif fonctionne effectivement comme une file, il est nécessaire que les places *Tête de File* et *Queue de File* possèdent le même marquage initial, un nom d'objet quelconque de la classe *ANY*

Pour illustrer davantage le fonctionnement de ce motif, et se convaincre qu'il fonctionne bien comme une file, nous allons lister l'évolution des marquages lors de séquences de tirs de transition, en vérifiant que seul les franchissements qui respectent l'ordre FIFO sont légaux.

Pour l'exemple, nous complétons la définition des places de la manière suivante :

Exemple :

Type

Any = ...

Personne = (Mathieu, Marc, Luc, Jean)

Places :

Tête_de_file, Queue_de_file : <a : ANY>;

File : <p : Personne, avant : ANY, après : ANY>

Le tableau ci-dessous liste l'évolution des marquages dans le cas de l'arrivée successive de Mathieu, Marc, Luc et Jean dans la place. Par commodité on notera les noms d'objets sous la forme *Classe#Numéro_d_instance* , ainsi *ANY#4* désigne la quatrième instance de la classe *ANY*

Transition	Marquage atteint		
	Queue de file	Tête de file	File
	<ANY#1>	<ANY#1>	
Enfiler <Mathieu>	<ANY#2>	<ANY#1>	<Mathieu, ANY#1, ANY#2>
Enfiler <Marc>	<ANY#3>	<ANY#1>	<Mathieu, ANY#1, ANY#2>, <Marc, ANY#2, ANY#3>
Enfiler <Luc>	<ANY#4>	<ANY#1>	<Mathieu, ANY#1, ANY#2>, <Marc, ANY#2, ANY#3>, <Luc, ANY#3, ANY#4>
Enfiler <Jean>	<ANY#5>	<ANY#1>	<Mathieu, ANY#1, ANY#2>, <Marc, ANY#2, ANY#3>, <Luc, ANY#3, ANY#4>, <Jean, ANY#4, ANY#5>
Défiler → <Mathieu>	<ANY#5>	<ANY#2>	<Marc, ANY#2, ANY#3>, <Luc, ANY#3, ANY#4>, <Jean, ANY#4, ANY#5>
Défiler → <Marc>	<ANY#5>	<ANY#3>	<Luc, ANY#3, ANY#4>, <Jean, ANY#4, ANY#5>

Figure II.17 - Evolution des marquages pour le motif File

On vérifie que les jetons sont bien capturés par la transition *Défiler* dans l'ordre où ils ont été traités par la transition *Enfiler*. D'autre part, pour que ce motif soit substituable à une place, il faudrait l'encadrer par une place d'entrée et une place de sortie, toutes deux de capacité 1, qui ne sont pas incluses dans la figure pour plus de simplicité.

b. Place Pile

De la même manière, une place gérée comme une pile (Premier Arrivé, Dernier Sorti ou LIFO) peut être émulée par le motif illustré en Figure II.18. Pour que le motif fonctionne comme attendu, il suffit que la place *Sommet* soit initialement marquée avec un nom d'objet quelconque.

Comme précédemment, nous illustrons l'évolution des marquages en supposant le même ordre d'arrivée des jetons :

Exemple :

Type

Any = ...

Personne = (Mathieu, Marc, Luc, Jean)

Places :

Sommet : <a : ANY>;

Pile : <p : Personne, devant : ANY, derrière : ANY>

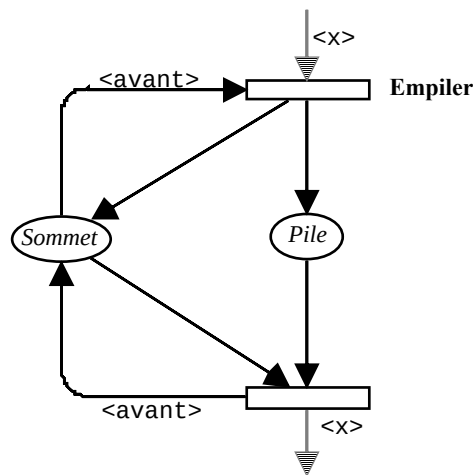


Figure II.18 - Un motif se comportant comme une Pile

Transition	Marquage atteint	
	Sommet	Pile
	<ANY#1>	
Empiler <Mathieu>	<ANY#2>	<Mathieu, ANY#1, ANY#2>
Empiler <Marc>	<ANY#3>	<Mathieu, ANY#1, ANY#2>, <Marc, ANY#2, ANY#3>
Empiler <Luc>	<ANY#4>	<Mathieu, ANY#1, ANY#2>, <Marc, ANY#2, ANY#3>, <Luc, ANY#3, ANY#4>
Empiler <Jean>	<ANY#5>	<Mathieu, ANY#1, ANY#2>, <Marc, ANY#2, ANY#3>, <Luc, ANY#3, ANY#4>, <Jean, ANY#4, ANY#5>
Dépiler → <Jean>	<ANY#4>	<Mathieu, ANY#1, ANY#2>, <Marc, ANY#2, ANY#3>, <Luc, ANY#3, ANY#4>
Dépiler → <Luc>	<ANY#3>	<Mathieu, ANY#1, ANY#2>, <Marc, ANY#2, ANY#3>

Figure II.19 - Evolution des marquages pour le motif Pile

Une place gérée en pile sera notée par le mot-clef **hard LIFO**.

c. Place triée

Pour exprimer qu'une place p est fortement triée suivant un critère Tri_p , nous allons décrire deux motifs différents. Le premier est une simple abréviation des RPO, mais nécessite de fournir explicitement les éléments de $\text{Type}(p)$ pour lesquelles la fonction Tri_p prend son maximum et son minimum⁴ ; le second n'a pas cet inconvénient, mais utilise un arc inhibiteur, qui en lui-même constitue une extension du pouvoir d'expression des RPO.

⁴ Le plus fréquemment, les fonctions Tri_p seront très simples, sinon triviales ; dans les exemples des Figures II.16 et II.18, la fonction Tri_p est l'identité, et les éléments à fournir seraient MinInt et MaxInt respectivement (limites de représentation des entiers, dépendantes de l'implémentation). Dans de tels cas les éléments pourraient bien entendu être calculés automatiquement.

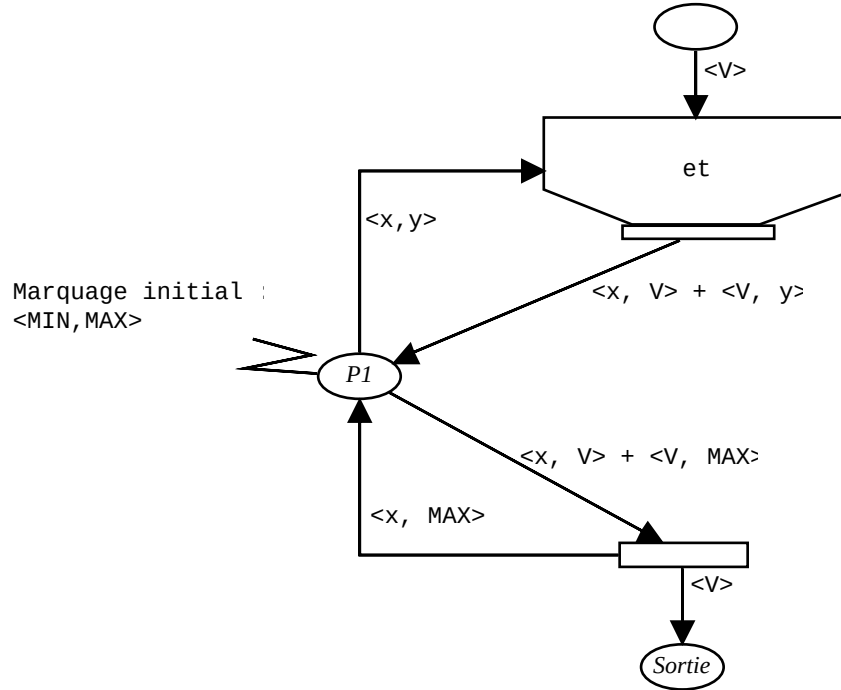


Figure II.20 - Motif se comportant comme une place triée (version 1)

Le motif décrit en Figure II.20 utilise un principe analogue au tri par insertion. On notera MIN (resp. MAX) la valeur pour laquelle la fonction Trip prend son minimum (resp. son maximum) sur son domaine.

Le marquage initial de la place P1 est constitué du seul doublet $\langle \text{MIN}, \text{MAX} \rangle$. Le principe qui guide l'évolution de ce motif est que, pour insérer le jeton V dans la place, on retire de cette place un doublet $\langle x_i, y_i \rangle$ tel $\text{Trip}(x_i) \leq \text{Trip}(V) \leq \text{Trip}(y_i)$, et on y insère les deux doublets $\langle x_i, V \rangle$ et $\langle V, y_i \rangle$.

Après quelques insertions le marquage de la place P1 est de la forme :

$$\{ \langle \text{MIN}, y_1 \rangle, \dots, \langle x_i, y_i \rangle, \dots, \langle x_n, \text{MAX} \rangle \} \text{ avec :}$$

$$\forall i = 1 \dots n-1 : \text{Trip}(x_i) \leq \text{Trip}(y_i) \text{ et } y_i = x_{i+1}$$

Lorsqu'on désire extraire de la place le jeton V qui, parmi les jetons insérés qui maximisent (resp. minimisent) la fonction Trip , il faut choisir deux doublets de la forme $\langle x_i, V \rangle, \langle V, \text{MAX} \rangle$ (resp. $\langle \text{MIN}, V \rangle, \langle V, x_i \rangle$), et y remettre le doublet $\langle x_i, \text{MAX} \rangle$ (resp $\langle \text{MIN}, x_i \rangle$). La Figure II.20 illustre donc un tri décroissant (on choisit la valeur qui maximise Trip).

Comme précédemment, nous illustrons le fonctionnement du motif par un exemple. On donnera ici $\text{Trip}(x) = \cos(x)$, et on a donc $\text{MIN} = \pi$, $\text{MAX} = 0$.

Le tableau ci dessous montre l'évolution du motif pour l'introduction successive de $\pi/2$, 0 et $3\pi/4$.

		Marquage atteint
Transition		P1
		$\langle \pi, 0 \rangle$
Entrer	$\langle \pi/2 \rangle$	$\langle \pi, \pi/2 \rangle, \langle \pi/2, 0 \rangle$
Entrer	$\langle 0 \rangle$	$\langle \pi, \pi/2 \rangle, \langle \pi/2, 0 \rangle, \langle 0, 0 \rangle$
Entrer	$\langle 3\pi/4 \rangle$	$\langle \pi, 3\pi/4 \rangle, \langle 3\pi/4, \pi/2 \rangle, \langle \pi/2, 0 \rangle, \langle 0, 0 \rangle$

Sortir $\rightarrow$ $\langle 0 \rangle$	$\langle \pi, 3\pi/4 \rangle, \langle 3\pi/4, \pi/2 \rangle, \langle \pi/2, 0 \rangle$
Sortir $\rightarrow$ $\langle \pi/2 \rangle$	$\langle \pi, 3\pi/4 \rangle, \langle 3\pi/4, 0 \rangle$

Figure II.21 - Evolution des marquages pour le motif Place triée

Nous avons décrit le cas où on trie des jetons d'arité 1. En toute généralité, si on désire trier par ce motif des jetons d'arité n dans une place, celle-ci sera d'arité $2n$.

On peut également utiliser les arcs inhibiteurs généralisés (§II.2.2.6) pour trier les jetons d'une place. Ceci est réalisé par le motif de la Figure II.22.

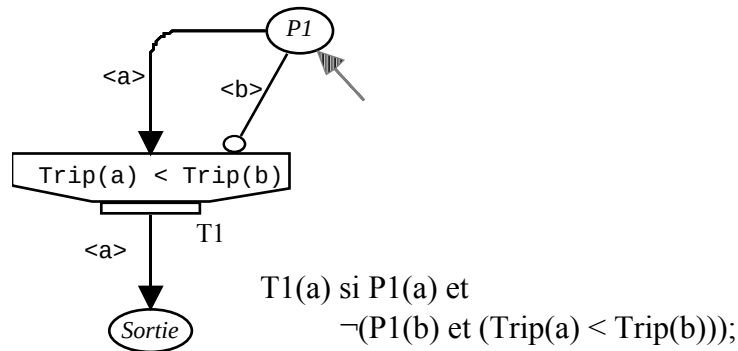


Figure II.22 - Motif se comportant comme une place triée (version 2)

Le motif de la Figure II.22 émule une place triée Hard Descending. On ne peut franchir la transition T1 avec un jeton $\langle a \rangle$ que s'il n'existe aucun jeton $\langle b \rangle$ dans la place tel que $Trip(a) < Trip(b)$. Un des intérêts de cette construction est que le tri est en quelque sorte "relatif à la transition". Plusieurs transitions peuvent donc "voir" la place triée suivant différents critères.

2.8. Prise en compte du temps

Les réseaux de Petri en général, et les RPO tels que nous les avons défini jusqu'à présent permettent d'exprimer la séquentialité, la précedence, la concurrence ou la synchronisation, et autorisent donc la définition de **liens de causalité** entre événements ; ils ne permettent de modéliser une dimension importante de l'évolution des systèmes, la **dimension temporelle**, que d'une manière qualitative et non pas quantitative.

Pour remédier à cette carence, et autoriser la prise en compte quantitative du temps dans les modèles, plusieurs modèles de RdP temporisés ont été proposés :

- Les **RdP temporels** [Merlin 74, Merlin...76] associent à chaque transition un couple $(t_{\min}, t_{\max})$ où $t_{\min}$ donne la durée minimale avant que la transition ne puisse être franchie, et $t_{\max}$ permet de calculer la date avant laquelle la transition armée doit effectivement être franchie. La date réelle de franchissement peut être n'importe quel point de l'intervalle ;
- Les **RdP à transitions temporisées** [Ramchandani 74] associent à chaque transition t_i du réseau une durée de tir r_i . Ce modèle identifie une transition à une activité dont r_i est la durée ;

- Les **RdP à places temporisées** [Sifakis 79] constituent l'approche duale de la précédente, où une durée r_i est associée à chaque place. Un jeton déposé dans la place p_i à la date u ne sera disponible pour le franchissement d'une transition qu'après la date $u + r_i$;
- Les **RdP à arcs temporisés** [Alla 87] associent le coefficient r_{ji} à chaque arc de $\text{Post}(p_i, t_i)$ permettant de spécifier que le temps de séjour dans la place p_i dépend de la transition qui a déposé le jeton ;
- Les **RdP à arcs temporels** [Bastide...89, Bako 90] associent, comme les RdP à arcs temporisés, le temps aux arcs du réseau, mais avec une sémantique différente. Dans ce modèle un intervalle de temps $[a_{ij}, b_{ij}]$ est associé à chaque arc de $\text{Pré}(p_i, t_i)$ signifiant ainsi qu'un jeton déposé dans la place p_i à la date u ne pourra être utilisé que pour un franchissement de la transition t_i se produisant dans l'intervalle $[u+a_{ij}, u+b_{ij}]$. [Bako 90] définit la temporisation au niveau des RdP simples, et les valeurs a_{ij} et b_{ij} sont donc des constantes. Dans le cadre des RPO, le développement logique de cette extension est de faire des bornes de l'intervalle des fonctions de la valeur du jeton manipulé.

Chacune de ces extensions peut être aisément incorporée dans la définition des RPO, afin d'autoriser une prise en compte du temps. Toutefois, pour intégrer l'aspect temporel dans le formalisme des Objets Coopératifs, nous présentons une technique différente, qui permet d'émuler les modèles de temporisation précédemment décrits.

a. Registre temporel

La bonne prise en compte des données valuées au sein des RPO rend très aisée la gestion de la variable temps dans les modèles. L'idée de base est de disposer, dans le réseau, d'un **registre** (cf § II.2.2) servant d'horloge et dont la valeur contient la date courante.

Pour disposer des opérations usuelles sur les dates et les durées, nous introduisons tout d'abord les domaines de valeur suivants :

- **Date** est un domaine qui nous permet de désigner un instant ponctuel, dont l'échelle et l'origine dépendent du système considéré;
- **Duration** est un domaine permettant de désigner des durées et des intervalles de temps, avec la même échelle que Date.

On dispose des opérateurs suivants :

+ : Duration X Duration → Duration

permettant de faire la somme de deux durées.

- : Duration X Duration → Duration

permettant de faire la différence entre deux durées.

+ : Date X Duration → Date

permettant d'ajouter une durée à une date.

- : Date X Duration → Date

permettant de soustraire une durée à une date.

- : Date X Date → Duration

permettant de calculer l'intervalle de temps séparant deux dates.

Les dates et les durées sont d'autre part munies de la relation d'ordre $\leq$.

Pour prendre en compte le temps dans un RPO, il suffit d'introduire dans ce réseau un **registre temporel** (Now, <Date>, 0). La manière dont ce registre évolue n'est pas définie dans le réseau (On suppose alors que c'est l'environnement qui met à jour la valeur de l'horloge). On peut également utiliser les techniques décrites dans [Richter 85, Oberweis 86] pour décrire l'évolution de l'horloge au sein même du réseau. En tout état de cause, la seule hypothèse faite est que l'évolution de cette valeur est monotone et croissante.

Conformément à la définition des registres (§ II.2.2), toute transition du réseau peut accéder à la valeur de Now, qui est interprétée comme la date courante au moment du franchissement de la transition. Bien entendu, comme l'évolution de la variable Now est extérieure au réseau, les transitions ne peuvent y faire que des accès en lecture (i.e. en partie droite des affectations).

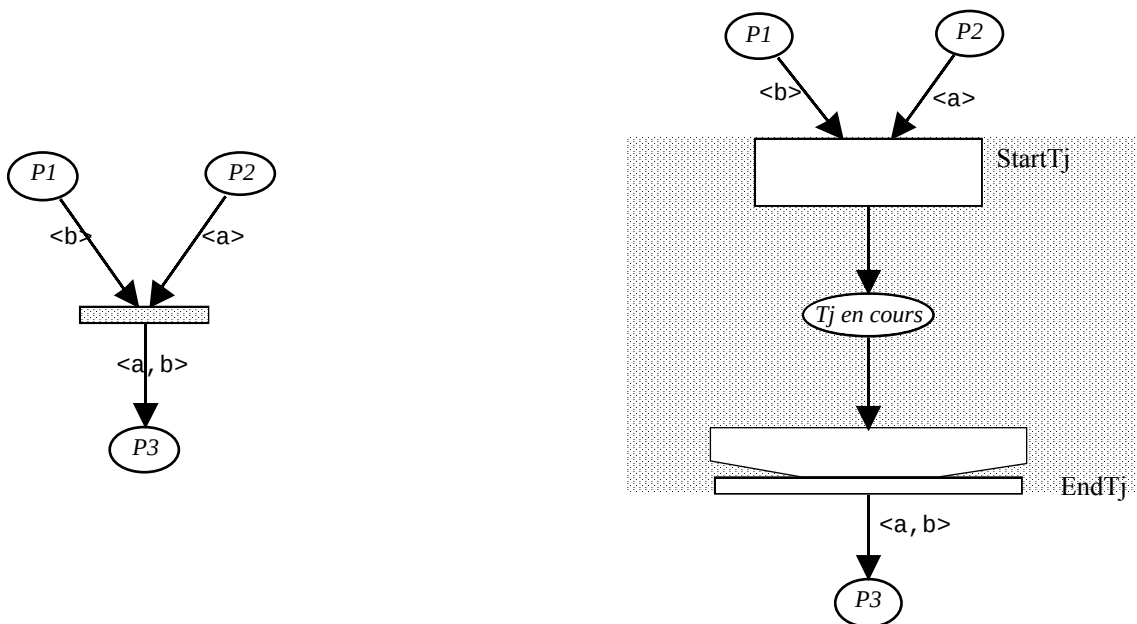


Figure II.23 - émulation d'une transition temporisée par un registre temporel

Cette extension très simple, combinée avec la puissance d'expression des préconditions et des actions et avec la possibilité d'assembler dynamiquement des tuples par occurrence de transitions, nous donne une grande flexibilité dans la manipulation du temps dans un réseau :

- La Figure II.23 montre comment émuler une transition temporisée "à la Ramchandani", par introduction d'une place intermédiaire représentant le fait que la transition originale est "en cours de franchissement" ;
- De même, les réseaux à arcs temporels peuvent être simplement émulés, comme le montre la Figure II.24, en enrichissant le type de la place d'un élément de type *Date* et en ajoutant des préconditions aux transitions de sortie. Cette figure illustre un cas d'utilisation classique des arcs temporels : la définition d'un **chien de garde** (watch-dog), c'est à dire d'une action de sauvegarde (ici la transition *Alarme*) qui doit être déclenchée si un traitement nominal n'a pu être effectué avant un certain laps de temps.

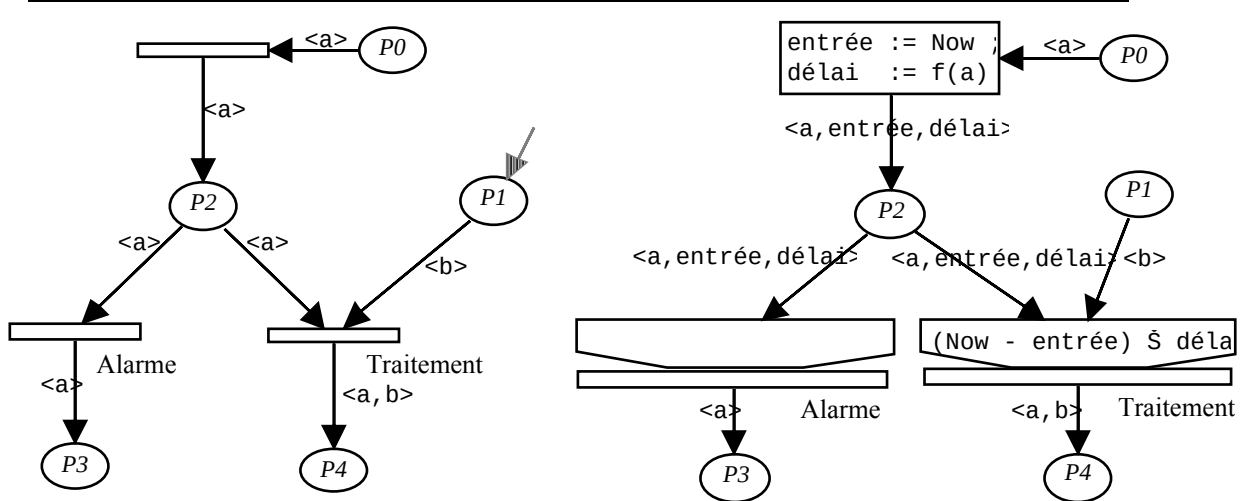


Figure II.24 - émulation des réseaux à arcs temporels par registre temporel

Dans les exemples des Figures II.23 et II.24, on peut également illustrer un apport intéressant de la technique du registre temporel par rapport aux définitions de base de Bako ou Ramchandani : il est très aisé de calculer des délais et des gardes en fonction de la valeur des jetons manipulés. La valeur de *délai*, qui est une constante dans ces deux figures, peut tout aussi bien être calculée en fonction de la valeur des jetons qui transitent ; en Figure II.23, par exemple, la transition StartTj pourrait contenir une action de la forme $délai := f(a, b)$. Une utilisation possible de ce mécanisme serait, dans le cadre de la modélisation des ateliers flexibles, d'exprimer le fait qu'une opération sur un poste de travail est fonction à la fois de la pièce manipulée et de la machine qui effectue cette opération.

La technique du registre temporel nous paraît très souple, et permet d'envisager des applications complexes de modélisation liée au temps : on pourrait par exemple prendre des décisions dans un réseau en fonction de la durée moyenne d'attente d'un jeton dans les places, ou du temps écoulé depuis son introduction dans le réseau. Cette technique permet également de modéliser très facilement des opérations périodiques (qui doivent être exécutées à intervalle de temps réguliers), ou des activités dont la date de début est fixée (il suffit de munir la transition d'une précondition de la forme $Now \bullet début$).

Du point de vue méthodologique, chaque domaine d'application du formalisme (application temps réel, modélisation d'un atelier en vue d'évaluation des performances, systèmes d'information,...) devrait définir une politique standard de prise en compte du temps, et éventuellement fournir les abréviations nécessaires.

Dans la troisième partie de ce mémoire, nous présentons une étude de cas portant sur la modélisation d'un atelier de production. Cette étude traite des aspects temporels du fonctionnement de l'atelier, et nous avons adopté pour la traiter la notation des arcs temporels, illustrée en Figure II.24.

3. STRUCTURATION DES MODELES DE RESEAUX DE PETRI DE HAUT NIVEAU

La difficulté principale que l'on rencontre lors de la modélisation d'un système par RdP de haut niveau est que l'on se trouve souvent confronté à des réseaux trop grands pour être aisément compréhensibles. La complexité de tels réseaux les rend difficiles à analyser, à modifier et à réutiliser. Le problème n'est pas différent de celui de l'explosion de la taille du logiciel, et comme dans ce domaine deux approches peuvent être suivies pour pallier ces problèmes :

- La première est la décomposition hiérarchique. Dans le domaine des RdP, cette approche consiste à substituer à une partie d'un réseau une autre partie plus détaillée, mais qui est considérée comme équivalente à la partie initiale, par une notion d'équivalence qui doit être précisée. Cette équivalence peut être d'ordre essentiellement graphique [Reisig 86], ou bien concerner explicitement le comportement du réseau. Si le comportement est pris en compte, la partie substituée peut être une transition [Valette 79] ou une place. Elle peut aussi consister en un sous-réseau, et l'équivalence est plus difficile à caractériser [Pomello 86]. Les techniques de réduction de Berthelot [Berthelot 86], qui visent à simplifier les réseaux, relèvent également de cette approche ;

Les limitations de la décomposition hiérarchique sont bien connues dans le domaine du logiciel, et les RdP n'y échappent pas : un système est toujours modélisé comme un seul réseau, et au niveau le plus détaillé il peut être très complexe. Ce réseau est un tout rigide, où le même sous-réseau peut se retrouver à différents endroits ; le concepteur a du mal à s'intéresser à une partie du système indépendamment des autres.

- La seconde approche n'imbrique pas des réseaux les uns dans les autres, mais au contraire considère des réseaux qui communiquent entre eux ; chacun de ces réseaux modélise alors une partie du système, et le système est le résultat de leur composition. La composition peut se faire par places partagées [Souissi...89] ou par transitions partagées [De Cindio...81]. L'activité inverse de la composition, qui consiste à décomposer un réseau en sous réseaux communicants a été également étudiée [Hack 72, Colom...86, Souissi...89] ;

Cette approche présente elle aussi des problèmes : chaque réseau composant doit être défini de telle sorte qu'il présente une unité fonctionnelle fortement cohérente et faiblement couplée, afin qu'il corresponde effectivement à un sous-système du système global. On souhaiterait de plus déduire le comportement ou les propriétés du système global à partir de celles de ses composants. Pour que cela soit possible, il est nécessaire que le protocole de communication entre les composants soit formellement défini.

Ces deux approches ne sont pas mutuellement exclusives, et toutes deux sont utiles au concepteur. La première permet de considérer un système à différents niveaux d'abstraction, et est bien adaptée à un processus de conception descendant, alors que la seconde permet de se concentrer sur différentes parties ou fonctions d'un système, et correspond mieux à un processus ascendant.

L'approche Objet est à ce jour la meilleure synthèse de ces deux approches : ses concepts permettent de considérer un système comme une composition de parties, qui sont tout d'abord considérées d'un point de vue abstrait. Ces parties sont ensuite conçues plus en détail, donnant lieu à des sous-systèmes sur lesquels il est possible d'itérer le même processus de décomposition, jusqu'à ce que l'on obtienne le modèle souhaité [Booch 87].

Nous allons dans la suite présenter trois propositions caractéristiques qui appliquent aux RdP respectivement la décomposition hiérarchique, la composition par réseaux communicants et la décomposition Orientée-Objet.

3.1. Approche par décomposition

Une des techniques de structuration des RdP par une approche de décomposition hiérarchique les plus abouties est celle présentée par Huber, Jensen et Shapiro [Huber...89], introduite pour structurer les modèles de RdP colorés.

Le but de cette approche est d'introduire dans les RdP des concepts comparables aux sous-modèles de SADT [Ross 79] ou aux sous-programmes des langages de programmation.

Le modèle d'intégration des sous-réseaux dans un réseau global n'est pas toutefois celui de l'appel de sous-programme, mais plutôt celui de la **substitution en ligne** de macro-instruction. Pour spécifier comment le sous-modèle spécifié s'intègre dans le réseau de niveau supérieur on définit des **nœuds de substitution** qui sont des macro-places ou transitions reliées aux sous-modèles, et la sémantique de substitution est définie de telle sorte que l'on puisse, à partir d'un modèle structuré hiérarchiquement, reconstruire automatiquement un RdP coloré non-hiérarchique équivalent (sauf pour un des modes de composition la **transition d'invocation**, que nous détaillons plus loin). Cette construction a pour conséquence que les modèles substitués doivent se comporter comme des places, ou comme des transitions.

Cinq constructions hiérarchiques sont proposées :

- Les **transitions de substitution**, où le nœud à substituer se comporte comme une transition. Le sous-réseau à substituer comporte un certain nombre de places d'interface (en entrée et en sortie) qui constituent en quelque sorte les paramètres formels de la substitution. Le réseau de niveau supérieur désignera parmi ses propres places celles qui doivent servir de paramètres réels à la substitution, par une notation simple, à la fois textuelle et graphique ;
- Les **places de substitution** sont la construction duale des transitions de substitution, où le réseau à substituer se comporte comme une place et offre une interface composée de transitions. Si plusieurs transitions du réseau de niveau supérieur font référence à la même transition d'interface, cette dernière doit être dupliquée autant de fois que nécessaire ;
- Les **places et transitions de fusion** sont essentiellement une facilité graphique, permettant de représenter graphiquement la même entité (place ou transition) à plusieurs endroits du réseau. Nous utiliserons cette facilité dans notre propre formalisme (cf §III.3) ;
- Enfin les **transitions d'invocation** ont pour but d'émuler l'appel de procédure des langages de programmation. Chaque appel (éventuellement récursif) implique une instanciation temporaire du sous-réseau correspondant (donc une modification dynamique de la structure du réseau), et

il est donc impossible de construire un RdP coloré équivalent. Bien entendu, avec une telle sémantique, on renonce pratiquement à toute possibilité d'analyse statique des RdP.

Un des avantages de cette approche est que les modèles produits sont plus simples et plus lisibles que ceux qu'on obtiendrait par les techniques classiques de "pliage" qui permettent de passer d'un modèle de RdP à un modèle de RdP colorés équivalent, mais plus concis. Ici, le pliage est remplacé par une substitution "en ligne" des sous-réseaux.

Nous donnons ici certains points permettant de comparer cette approche à la nôtre :

- Dans les Objets Coopératifs, on épargne également au concepteur la tâche de "replier" lui-même ses réseaux pour pouvoir gérer plusieurs instances du même processus dynamique. Toutefois, le pliage est obtenu automatiquement et **sans duplication des sous-réseaux** ;
- Il est possible dans les Objets Coopératifs d'obtenir des comportements dynamiques (et même récursifs) comparables à ceux décrits par les transitions d'invocation. Toutefois nous obtenons ces comportements avec un réseau global **dont la structure n'évolue pas**. La technique permettant cela (décrite au §V.3) serait d'ailleurs parfaitement applicable aux transitions d'invocation ;
- Enfin, les auteurs évoquent la possibilité d'avoir «des couleurs de jetons qui référencent d'autres réseaux colorés» mais soulignent que «pour le moment» cette technique n'a pas été retenue. D'une certaine manière, c'est cette possibilité d'avoir des jetons qui référencent d'autres RdP qui est à la base de notre formalisme.

La discussion ci-dessus est donnée sans justification. Tous ces points seront abondamment repris et détaillés au chapitre V, qui traite de la sémantique formelle de notre formalisme.

3.2. Réseaux communicants

Parmi les approches les plus caractéristiques des approches de composition des RdP nous détaillons ici celle de Y.Souissi et G.Memmi [Souissi...89].

Les objectifs de cette approche sont de trouver des techniques de composition et de décomposition des RdP qui préservent le comportement (et notamment les "bonnes" propriétés), afin de permettre la construction modulaire d'un système et de déduire ses propriétés par l'analyse de ses composants élémentaires.

L'originalité de l'approche est de considérer qu'une communication entre RdP implique en fait trois réseaux et non pas deux, à savoir les deux réseaux qui communiquent, et le médium de communication, lui-même représenté comme un réseau. On recherche ensuite comment les propriétés (telles que la vivacité) se conservent, depuis les réseaux qui communiquent jusqu'à celui qui résulte de leur communication, en fixant des contraintes structurelles uniquement sur le médium.

Quatre formes de médium de communication sont étudiées : par places (communication unidirectionnelle), par rendez-vous, par processus séquentiel, ou par bloc bien formé :

- **Communication par places partagées** : Le médium de communication est constitué uniquement de places, et on suppose de plus que l'un des réseaux communicant n'a aucune place d'entrée dans l'ensemble des places partagées. Dans ce cas, il est montré que la vivacité est préservée, à savoir que les deux propositions suivantes sont équivalentes :
 - Les sous réseaux communicants sont vivants ;
 - Le réseau résultant de la communication est vivant.

De plus un algorithme est donné pour calculer toutes les **frontières** (séparation lines) dans un RdP afin de le décomposer en sous-réseaux communiquant par places partagées. Bien entendu, dans un réseau complexe, il existera plusieurs décompositions possibles, et il est probable qu'une seule de ces décompositions donnera des sous-réseaux présentant une unité fonctionnelle ;

- **Communication par processus séquentiel** : les processus séquentiels sont ici définis comme une extension de ceux de [Reisig 82]. Si le médium est un processus séquentiel (ou un **rendez-vous** qui en est un cas particulier) des résultats de conservation de la vivacité et du caractère borné des réseaux sont obtenus ;
- Enfin, dans le cas de la **composition par blocs bien formés**, les résultats de [Valette 79] sont repris et étendus pour montrer la préservation de la vivacité.

Nous avons repris dans le formalisme des Objets Coopératifs l'idée intéressante de définir par un RdP le médium de communication entre les objets (cf §III.4). Dans notre cas, le médium est uniquement composé de places, mais les résultats de [Souissi...89] ne sont pas applicables, car nos objets ont une communication de type question/réponse, et donc bi-directionnelle.

3.3. Structuration Orientée-Objet : La méthode HOOD/PNO

L'intégration des approches Objet et Réseaux de Petri est un domaine où la recherche est très active, et certains formalismes ont déjà été proposés (par exemple [Bruno...86]). Nous présentons ici la méthode HOOD/PNO [Paludetto 91] dont les concepts sont les plus proches de ceux de notre propre formalisme.

La méthode HOOD/PNO est une méthode de développement de logiciel, dont le domaine d'application principal est le développement de logiciel de commande en temps réel de procédés industriels. Elle propose un cycle de vie complet, allant de l'analyse jusqu'à la production de code, en passant par la conception architecturale et détaillée.

Il s'agit d'une méthode "toute Objet", où toutes les phases du cycle de vie justifient du même formalisme, permettant ainsi une transition facile entre étapes et une trace aisée des entités au cours du développement.

En termes de conception architecturale, la méthode HOOD/PNO s'appuie sur les concepts de la méthode HOOD (cf §I.3), l'idée directrice étant d'utiliser les RdP pour la description formelle des ObCS et des OpCS.

Cette utilisation systématique des RdP permet de pallier aux défauts les plus importants de la méthode HOOD, qui sont dus au manque de formalisation des ObCS et OpCS. Dans la méthode HOOD, on recommande de décrire ObCS et OpCS par les constructions des tâches Ada. En suivant cette recommandation, on s'expose à plusieurs inconvénients :

- On fixe tout d'abord de façon prématurée le langage de programmation, car des constructions équivalentes aux tâches sont peu souvent disponibles dans d'autres langages ;
- On utilise un outil dont la définition formelle est pour le moins insuffisante (que l'on songe, par exemple, aux problèmes liés à la terminaison des tâches imbriquées...), ce qui compromet les possibilités de raisonner sur une conception ou de la valider ;
- On travaille enfin avec un modèle d'exécution séquentiel, dans le cadre duquel il est difficile de décrire des entités au comportement concurrent. On court alors le risque de rechercher trop tôt les processus séquentiels qui composent le système.

L'utilisation des RdP permet également d'éviter la centralisation du contrôle d'un objet parent dans un seul de ses objets enfants (CTRL_parent, cf §I.3), qui conduirait à mettre en place une dépendance verticale entre les objets d'une hiérarchie. On contraire, on distribuera en général le contrôle d'un parent de manière équitable entre ses enfants.

La méthode HOOD/PNO met à profit l'utilisation des RdP pour guider le concepteur, dans son processus de modélisation, pour la découverte de nouveaux objets. Après avoir définie par un RdP le comportement d'un objet de niveau n , l'analyse des invariants linéaires des places de son ObCS permet de déterminer des bons candidats comme ObCS des objets de niveau $n+1$. Cette utilisation d'une technique d'analyse formelle au sein même d'un processus de conception est particulièrement remarquable, car les techniques couramment proposées pour la découverte des objets [Booch 87, Meyer 90b] sont le plus souvent d'ordre empirique, et ce problème préoccupe particulièrement les ingénieurs logiciels qui souhaitent suivre une démarche Orientée-Objet.

Enfin, la méthode se poursuit jusqu'au niveau de la production de code, en définissant des règles de traduction semi-automatique, en traduisant par des structures de contrôle Ada les ObCS des objets de la conception.

Les principes fondamentaux de la méthode HOOD/PNO sont comparables à ceux de notre approche, et nous avons personnellement eu de nombreuses réunions de travail avec M.Paludetto et les autres membres du groupe IPANEMA, conduit par Robert Valette, où certaines de ces idées ont vu le jour. Pour situer plus précisément notre contribution par rapport à la sienne, citons les points suivants :

- HOOD/PNO est une méthode complète, qui couvre tout le cycle de vie, et est plus particulièrement orientée vers la commande de procédé. Les Objets Coopératifs sont simplement un formalisme (ou peut être un langage de programmation, voir à ce sujet la conclusion de ce mémoire) ; à ce titre, notre objectif a été de le rendre le plus général possible, afin d'étendre au maximum son domaine d'application. Nous avons donné peu d'indications méthodologiques, considérant que chaque domaine d'application relève de sa propre méthode de conception ;

- HOOD/PNO conserve le modèle de structuration de HOOD, qui est extrêmement statique. Nous avons introduit dans les Objets Coopératifs les constructions dynamiques rencontrées dans la plupart des langages Orientés-Objet, avec notamment l'instanciation dynamique et la relation d'utilisation dynamique entre objets ;
- Enfin les Objets Coopératifs intègrent complètement les mécanismes liés à la classification et à l'héritage, gérant notamment l'héritage multiple et le polymorphisme, concepts qui sont absents de la méthode HOOD. Nous pensons que l'héritage (lorsqu'il a la sémantique d'une spécialisation ou **sous-typage** et non pas d'une extension, cf §I.1.4) est un outil conceptuel extrêmement important dans un processus de conception (l'utilisation de l'héritage en tant que mécanisme d'extension, par contre, relève essentiellement de l'activité de conception détaillée ou de codage).

PARTIE II :

LES OBJETS COOPERATIFS

Chapitre III. Définition des objets coopératifs

Le formalisme des Objets Coopératifs est un langage de conception Orienté Objet, qui permet la modélisation d'un système à des niveaux d'abstraction différents en le décrivant comme une collection d'objets concurrents, qui coopèrent d'une manière précisément définie afin de remplir les services attendus du système.

En paraphrasant une équation célèbre, notre approche peut être caractérisée par :

Système = Objets + Coopération
Objet = Structure de Données + Opérations + Comportement

Figure III.1 - Equation fondamentale du modèle

L'approche objet nous fournit les concepts suffisants pour définir la structure des objets et leurs inter-relations, afin de structurer le système suivant les principes de forte cohésion et de faible couplage ; la théorie des réseaux de Petri, quant à elle nous permet de modéliser le comportement des objets et les communications qu'ils entretiennent, de telle sorte que l'on puisse exprimer la concurrence aussi bien entre les différents objets qu'au sein même d'un objet.

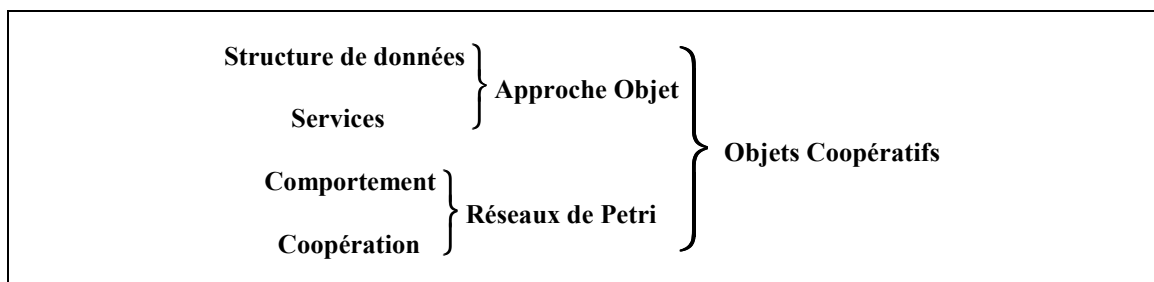


Figure III.2 - Le formalisme des Objets Coopératifs et ses formalismes de référence

Pour ce qui est des composants "structure de données" et "services", l'approche objet est aujourd'hui suffisamment riche pour que l'on y trouve facilement les concepts les mieux adaptés à nos objectifs. Un objet sera donc défini comme une instance de sa classe, et pourvu d'un ensemble d'attributs et d'un ensemble de services (ou méthodes) qui permettent la manipulation de ses attributs. Cette définition est faite d'une manière très classique, proche du langage Eiffel.

La structure d'un objet est enrichie par l'ajout d'un ObCS (Object Control Structure, §I.3.6) qui définit sa structure de contrôle interne, ou son **comportement**. Les ObCS sont décrits par des Réseaux de Petri à Objets (§II.1.2), et permettent de décrire la disponibilité des services en fonction de l'état interne de l'objet, et les changements de cet état produits par l'exécution des services.

Le rôle d'un objet dans un système est d'aider les autres objets en mettant à leur disposition ses services, et en accomplissant les demandes d'exécution de services (les **invocations**) qu'il reçoit. Un

objet peut ainsi être considéré du point de vue de ses **clients** (les objets pour le compte desquels il exécute des opérations) ou du point de vue de ses **serveurs** (les objets qu'il invoque).

La définition d'un objet du point de vue de ses clients est sa **spécification**. Elle inclut ceux parmi ses attributs qui sont accessibles en lecture par d'autres objets, les opérations qui peuvent être invoquées et un **ObCS de spécification** qui indique les séquences d'invocations que l'objet est à même d'accomplir.

L'**implémentation**, quant à elle, décrit comment l'objet est effectivement constitué, et précise comment l'objet se comporte lui-même en temps que client, en invoquant d'autres objets.

La **sémantique de la communication** entre objets est décrite en restant dans le cadre formel de la théorie des réseaux de Petri, ce qui permet de caractériser formellement le comportement d'un système d'objets qui communiquent, le comportement de chacun des objets du système étant lui-même décrit par un RdP.

Ce chapitre est consacré à la définition des classes d'Objets Coopératifs, aussi bien en ce qui concerne la spécification que l'implémentation.

Le chapitre IV décrit la structure d'un système d'Objets Coopératifs, et les techniques de structuration de l'espace des classes.

Le chapitre V est consacré à la description de la sémantique formelle d'un système d'Objets Coopératifs et le chapitre VI décrit des extensions du formalisme, destinées à en faciliter l'usage ou à étendre son domaine d'application.

1. PRESENTATION DU FORMALISME

Le formalisme des Objets Coopératifs est un langage de conception Orienté Objet, qui présente les caractéristiques suivantes :

- C'est un langage à **base de classe**, qui force à opérer une distinction claire entre le concept de classe et celui d'instance. Une classe est la description statique de la structure et du comportement des objets instances de cette classe, qui seront créés dynamiquement. Dans le formalisme des Objets Coopératifs, l'unité de conception est la classe d'objets. L'objet, instance d'une classe, est le composant de base du système ;
- Il favorise l'**abstraction** en donnant la possibilité de décrire séparément la spécification d'une classe d'objets et son implémentation, tout en autorisant une vérification formelle de la cohérence de ces deux descriptions ;
- Il permet le masquage d'information et l'**encapsulation**, en définissant pour une classe des opérateurs d'accès qui cachent la structure de donnée physique des instances ;
- C'est un langage **fortement typé** : le type de chaque entité (variable, constante, paramètre,...) du langage fait partie de sa définition, et ce type détermine son domaine de valeur. Le type d'une expression quelconque du langage peut être déterminé de manière statique au simple examen du texte de la classe dans laquelle cette expression est définie ;
- C'est un langage à **sémantique de référence** : la manipulation des objets se fait par l'intermédiaire de leur référence (on parle aussi de nom des objets), et des opérations telles que l'affectation ou la comparaison portent sur les références et non sur la valeur des objets référencés ;
- C'est un langage à **création dynamique** : les objets peuvent être créés dynamiquement dans le système au cours de son activité ;
- C'est un langage qui autorise le **polymorphisme**, c'est à dire la possibilité pour un identifieur du langage d'être lié à un objet d'une classe dérivée par héritage de la sienne propre. L'héritage introduit entre les classes une relation de **compatibilité**, qui indique sous quelle condition une valeur d'un certain type peut être affectée à une variable d'un type différent. Cette notion de compatibilité entre classes permet de relâcher de manière contrôlée l'aspect fortement typé du langage.

Les sous-chapitres III.2 à III.4. constituent en quelque sorte le "manuel de référence" du langage des Objets Coopératifs, décrivant à la fois ses aspects syntaxiques et sémantiques. Les deux sous-chapitres qui suivent la définition du système de types en sont les plus importants : ils définissent comment une classe d'Objets Coopératifs est décrite suivant le point de vue externe (sa spécification, §III.3) et interne (son implémentation §III.4).

2. LE SYSTEME DE TYPES

Le formalisme des objets coopératifs fait partie de la familles des langages fortement typés : le type de chaque entité (variable, constante, paramètre,...) du langage fait partie de sa définition, et l'entité ne pourra être liée qu'à des valeurs de ce type. Le type d'une expression quelconque du langage peut être déterminé de manière statique au simple examen du texte de la classe dans laquelle cette expression est définie.

Le système de types proposé par le langage distingue les **types simples** des **types classe** :

2.1. Types simples

Le domaine de valeurs d'un type simple est un ensemble de constantes, et une variable de type simple aura donc pour valeur une constante. Les types simples sont soit des **types prédéfinis**, soit des **types construits**.

Les types prédéfinis sont les suivants :

- *INTEGER* qui dénote les valeurs entières signées ;
- *REAL* qui dénote des valeurs signées à virgule flottante ;
- *BOOLEAN* qui dénote des valeurs booléennes, parmi les constantes *TRUE* et *FALSE* ;
- *CHAR* qui dénote des valeurs de type caractère ;
- *STRING* qui dénote des chaînes de caractères.

Les opérations que l'on rencontre usuellement dans les langages de programmation pour ces types prédéfinis (affectation, comparaison, traitements numériques) sont également prédéfinies dans le langage.

En ce qui concerne le type *STRING*, les opérations classiques sur les chaînes seront données en utilisant la notation pointée *variable.opération(liste de paramètres)* et en utilisant les opérateurs de manipulation de chaîne fournis par le langage Eiffel. Il faut toutefois remarquer que notre type *STRING* n'est pas identique à la classe *STRING* du langage Eiffel [Meyer 90b] : en effet des éléments de ce type seront toujours manipulés par valeur.

Les types construits sont soit les types définis par intervalle ou énumération, comme dans le langage Pascal, soit des types créés par application d'un **constructeur de type** à des types déjà connus. Deux constructeurs de type sont définis, en utilisant la syntaxe du langage Pascal.

- *set of* qui construit un type simple dénotant un ensemble de valeurs de type quelconque. Les opérations ensemblistes usuelles sont définies pour les valeurs des types ensemblistes ;

- *array of* qui construit un type simple dénotant un tableau de valeurs de type quelconque. Les indices d'un tableau prennent leur valeur dans un type défini par intervalle ou par énumération, et chaque élément du tableau est accessible par son indice.

2.2. Types classe

Les types classe permettent aux identificateurs du langage de dénoter non plus une constante mais un **objet** du système. Une variable dont le type est une classe aura pour valeur le **nom d'un objet** de cette classe. Une variable de type classe est une référence vers l'objet qu'elle dénote.

Quatre opérations sont définies pour un identificateur de type classe :

- L'**affectation**, qui permet d'associer un nouveau nom d'objet à l'identificateur. L'opérateur correspondant est `:=` comme pour l'affectation entre variables de type simple ;
- Le **test d'identité**, qui permet de tester si deux références d'objets sont égales. Cet opérateur est noté `=` comme pour le test d'égalité entre variables de type simple ;
- L'**invocation**, qui désigne l'action de demander l'exécution d'un service à l'objet dénoté par l'identificateur. La syntaxe et la sémantique de l'invocation sont définies précisément en §VI.1.

La différence fondamentale entre types simples et types classe doit être soulignée : une variable de type simple est liée à une valeur de ce type, et les opérations portant sur cette variable ont une sémantique de valeur. Une variable de type-classe, au contraire, est liée au nom d'un objet de cette classe, et les opérations portant sur cette variable ont une sémantique de référence : l'affectation entre deux variables de type classe, par exemple, provoque une homonymie dynamique sur l'objet référencé⁵. Il faut noter également que les types construits sur des types classes sont toujours des types simples, et sont donc manipulés par valeur.

3. SPECIFICATION D'UNE CLASSE

La spécification d'une classe d'Objets Coopératifs fournit aux clients de cette classe toute l'information nécessaire à son bon usage. Nous nous appuyons, pour la définition d'une spécification et la signification que nous donnons à ce terme, sur la discussion du §I.1.2. Conformément à cette discussion, la spécification d'une classe va définir son **interface** et sa **pragmatique**. L'interface est décrite par une syntaxe inspirée du langage Eiffel, et la pragmatique de la classe est décrite en utilisant le formalisme des RPO.

3.1. Interface d'une classe

Le formalisme des Objets Coopératifs s'appuie clairement sur une organisation fondée sur une coopération de type client / serveur entre les entités du système.

⁵cf. [Meyer 90], pp. 114-118 pour une discussion détaillée sur les sémantiques de valeur et de référence dans les langages de programmation

Cette vision asymétrique de la communication entre objets se retrouve dès la définition de l'interface : l'interface d'une classe d'Objets Coopératifs doit fournir l'ensemble des informations nécessaires à la bonne utilisation d'une instance de cette classe ; les informations contenues dans l'interface d'une classe sont donc pertinentes pour les clients potentiels de cette classe, c'est à dire les autres classes pouvant être amenées à utiliser les services qu'elle **offre**.

L'interface doit fournir toutes ces informations, et seulement ces informations : il nous paraît dangereux d'inclure dans la spécification des informations d'une autre nature, comme par exemple la liste des services ou des classes **requises**, comme cela est proposé dans HOOD (cf §I.3). L'introduction d'une telle information va à l'encontre de plusieurs principes fondamentaux :

- La distinction spécification / implémentation : il est clair que l'interface requise est dépendante de l'implémentation, et l'on peut fort bien fournir pour une classe plusieurs implémentations radicalement différentes, qui ne requerront pas les mêmes services de leur environnement ;
- L'encapsulation ou la dissimulation d'information : les services que requiert une classe n'ont pas d'intérêt pour ses clients, et aucune décision concernant l'implémentation des clients ne devrait être prise au vu de l'interface requise par un de ses serveurs, puisque cette interface est susceptible de changer ;
- La réutilisabilité : si l'interface d'une classe inclut les informations requises en plus des informations offertes, il devient impossible de réutiliser une classe en dehors de son propre contexte.

L'interface d'une classe d'Objets Coopératifs est constituée d'un ensemble de types, d'un ensemble de constantes, d'un ensemble de **services** et d'un ensemble d'**attributs**, chacun de ces ensembles pouvant être vide. On parlera des types, constantes, services et attributs **exportés** ou **publiés** par la classe.

L'interface d'une classe contient tout d'abord des définitions de types : ces types sont utiles pour pouvoir définir des domaines particuliers pour les paramètres en entrée et en sortie des services. Pour chaque type exporté on donne :

- Son nom, dont la visibilité s'étend à toute classe cliente de celle où le type est défini ;
- Sa construction : on peut décrire explicitement comment le type est constitué à partir des constructeurs de base, et une classe cliente aura accès à tous les opérateurs légaux sur ce type. On peut également, comme en Ada, restreindre cet accès aux seules opérations définies sur tous les types simples, à savoir l'affectation, le test d'égalité et l'utilisation en tant que paramètre effectif, cette restriction étant alors spécifiée par le mot clé **is private**. Un type classe C_2 peut être publié en mode *private* dans l'interface d'une autre classe C_1 , empêchant ainsi les clients de C_1 d'effectuer des invocations sur les paramètres de classe C_2 .

L'interface contient ensuite la déclaration de constantes symboliques. Une constante est définie par :

- Son nom, qui à la même visibilité que les types exportés ;
- Son type qui est soit un type simple prédéfini, soit un type simple exporté par la classe (il n'existe pas de constantes de type classe).

Les constantes symboliques sont utiles pour désigner des valeurs singulières qui peuvent être renvoyées par les services.

Un service est défini par :

- Son nom, qui identifie complètement le service à l'intérieur de la classe. Contrairement à d'autres langages (tels que C++) on n'autorise pas la surcharge des noms de service, qui permet de fournir plusieurs versions du même service, qui est alors identifié par son nom et sa signature⁶ ;
- Sa signature, c'est à dire la définition de ses paramètres d'entrée et de sortie. Tous les services sont fonctionnels : ils ont une liste de paramètres en entrée , et une liste de paramètres en sortie, chacune de ces deux listes pouvant être vide. Il n'existe pas de services procéduraux, sans valeur de retour : on considère intuitivement qu'un service renvoie au minimum un **accusé d'exécution**, informant le client que le traitement associé au service est terminé. Une liste de paramètres vide (en entrée ou en sortie) est implicite et n'est donc pas mentionnée dans la signature ;
- Chaque paramètre d'entrée ou de sortie du service est défini avec un nom et un type, qui peut être un type simple prédéfini, un type exporté ou un type classe ;
- Le mode de passage des paramètres d'entrée d'un service est toujours le passage par valeur, et il n'est donc pas nécessaire de le préciser. Il faut bien entendu noter qu'une variable de type classe est liée à la référence d'un objet ; si une telle variable est passée en paramètre, sa valeur (le nom de l'objet qu'elle dénote) ne changera pas, mais l'objet dénoté pourra lui même être invoqué, et une invocation peut donc provoquer un effet de bord sur l'objet dénoté par un paramètre de type classe.

L'interface publie également la spécification de l'opération *create*, qui permet de créer dynamiquement de nouvelles instances. Cette opération fait l'objet d'un paragraphe spécial ci-après, en §III.3.3.

La classe exporte enfin un ensemble d'attributs, défini, comme les paramètres d'un service, par un nom et un type. Ces attributs sont accessibles **en lecture seulement**, à l'extérieur de la classe.

- Un attribut de type simple est appelé une **propriété** ;
- Un attribut de type classe est appelé une **référence**.

Il nous faut faire une remarque d'ordre méthodologique quant à la possibilité pour une classe de rendre publics des attributs dans sa spécification : cette possibilité n'est pas essentielle, et certains langages à objet (tel Smalltalk) s'en dispensent complètement. L'accès aux attributs pose plusieurs problèmes :

- D'une part, dans un contexte de programmation concurrente, l'accès à un attribut constitue un couplage bien plus intime que l'invocation, qui peut être implantée par passage de messages entre processus. Nous verrons au §V.3, que la sémantique de l'accès à un attribut, décrite en

⁶ Cette restriction n'est pas liée aux fondements théoriques du formalisme, et on pourrait envisager de fournir la possibilité de surcharge comme facilité syntaxique.

termes de RdP, est très différente de celle de l'invocation. Il y a donc lieu de s'interroger sur l'opportunité de publier des attributs si l'objet doit être faiblement couplé avec son environnement (par exemple implanté sur un ordinateur distant) ;

- D'autre part la publication d'un attribut a des répercussions importantes sur l'organisation de l'héritage, qui est discutée au §IV.2 : les attributs publiés par une classe ancêtre doivent être également publiés par toute classe dérivée ; en publiant un attribut, on contraint donc de manière très importante la structure de données, et donc l'implémentation de la structure physique de stockage, de toute classe dérivée, alors que l'héritage, lorsqu'on lui donne la sémantique du sous-typage, devrait contraindre uniquement la spécification d'une classe, et donc ses services (ceci rejoint la discussion du §I.1.4, sur la sémantique de l'héritage dans les LOO).

Malgré ces inconvénients, nous avons choisi de conserver la possibilité de donner accès à des attributs dans la spécification d'une classe, car elle nous paraît utile et justifiée dans certains cas, dont certains sont commentés en détail dans ce mémoire (cf §VII.1, §VI.4, §V.3), et dont d'autres font l'objet de recherches parallèles aux nôtres [Sibertin 91]. Au contraire, une démarche méthodologique axée sur d'autres domaines d'application (tel que la programmation répartie) pourrait proscrire totalement l'usage des attributs publics.

```

Class Specification Exemple;
Types
  Type1 : set of INTEGER;
          -- Type1 est un type construit exporté
  Type2 is private;
          -- Type exporté en mode privé
Constants
  Ct1, Ct2 : Type2;
          -- Deux constantes symboliques de type Type2
Services
  Sv1 <p1 : INTEGER, p2 : STRING> : <v1 : STRING, v2 : Type2>;
          -- le service Sv1 a deux paramètres d'entrée :
          -- p1 de type entier et p2 de type chaîne.
          -- Il a deux paramètres de sortie, v1 de type chaîne,
          -- et v2 de type Type2, type exporté

  Sv2 : <v1 : Type1>;
          -- service sans paramètre d'entrée, et avec un paramètre
          -- de sortie de type Type1, type exporté

  Sv3;
          -- service sans paramètre d'entrée ni de sortie
Attributes
  att1 : Type1;
  att2 : Set of INTEGER;

```

Figure III.3 - Syntaxe de l'interface d'une classe

La syntaxe de déclaration des types, des constantes, des services et attributs qui constituent l'interface d'une classe est illustrée en Figure III.3. On remarque notamment que les listes de paramètres d'entrée et de sortie sont délimitées par des signes "inférieur" et "supérieur" (< >) au lieu des parenthèses classiquement utilisées dans les langages de programmation. Cette syntaxe particulière sera justifiée

au §III.3.2. La figure illustre également quelques simplifications syntaxiques utilisées pour désigner des fonctions dont la liste de paramètres d'entrée ou de sortie est vide.

3.2. Pragmatique d'une classe

Pour compléter la spécification d'une classe d'objets, il convient d'expliciter la **pragmatique** des objets de cette classe. Il faut fournir une description de l'objet qui explicite son mode d'emploi, en détaillant notamment sa structure de contrôle, c'est à dire :

- Les conditions d'activation de ces services, et plus particulièrement :
 - * les séquençements possibles d'activation des services offerts ;
 - * les contraintes d'exclusion entre ces activations.

Cette description doit indiquer à la fois l'influence de l'état de l'objet sur la disponibilité de ses services, et l'influence de l'exécution des services sur l'état de l'objet. Au stade de la spécification, cette vision de l'interaction entre état de l'objet et exécution des services reste nécessairement abstraite, puisque la structure précise de l'objet n'est pas définie.

Dans le formalisme des Objets Coopératifs, la pragmatique d'une classe est décrite en définissant pour ses instances une Structure de Contrôle d'Objets (par référence à la méthode HOOD, on gardera la dénomination anglaise d'**Object Control Structure** et on parlera d'**ObCS de spécification**). Cet ObCS est décrit par un Réseau de Petri à Objets.

L'ObCS d'une classe est mis en relation avec son interface de la manière suivante :

- Chaque service défini dans l'interface de la classe est associé à une transition au moins du RPO de l'ObCS par la fonction **Disp** (dite fonction de disponibilité) qui à chaque service fait correspondre un ensemble (non vide) de transitions. Une transition associée à un service, appelée **Transition de Service (TS)**, est graphiquement mise en évidence par un arc d'entrée et un arc de sortie qui ne sont reliés à aucune place du réseau (on parle d'arcs pendants) ; ces arcs seront appelés respectivement **arc d'activation** et **arc de complétion**. Par la suite, on désignera sous le nom d'**ObCS d'une classe** le couple constitué par le RPO et la fonction Disp ;
- L'exécution d'un service par l'objet équivaut au franchissement d'une des TS associées à ce service : lorsqu'un objet reçoit une invocation, il ne peut l'exécuter que si l'une des TS associées à ce service est franchissable dans le RPO. Lorsque cette invocation est effectivement exécutée, la TS correspondante est franchie, et le marquage du réseau est modifié en conséquence ;
- La liste des paramètres d'entrée d'un service (éventuellement vide) est indiquée sur l'arc d'activation de chaque transition associée. De même la liste des paramètres de sortie est reportée sur son arc de complétion. Ce report explique pourquoi on a choisi d'utiliser les délimiteurs "< >" dans la syntaxe de définition des listes de paramètres et de valeurs de retour des services : ces délimiteurs sont classiquement utilisés dans la représentation graphique des n-uplets de variables dans les réseaux Prédicats / Transition, et ici, on identifie la

liste de paramètres d'entrée (resp. de sortie) d'un service avec un n-uplet de variables d'entrée (resp. de sortie) de chaque transition associée ;

- L'ObCS peut contenir des transitions qui ne sont pas associées à un service ; ces transitions, appelées **Transitions Privées** (TP), servent à modéliser des changements d'état internes à l'objet, qui ne sont pas directement déclenchés par des invocations.

La définition d'un ObCS de spécification doit se restreindre à un sous-ensemble de la syntaxe définie au chapitre (II.1.2). Les restrictions sont les suivantes :

- Les attributs publiés par la classe sont utilisés comme registres de l'ObCS (cf §II.2.2). La définition du RPO ne doit pas comporter d'autres registres (on verra au §III.4 que cette restriction est levée lors de la définition de l'**implémentation** de la classe) ;
- Les actions associées aux transitions de l'ObCS ne doivent pas contenir d'invocation ; utiliser une invocation reviendrait en effet à préciser un des services **requis** par les instances de la classe dès sa spécification, pratique que nous avons rejetée. Une action sera donc constituée uniquement d'affectations et d'expressions simples des variables d'entrée de la transition.

Syntaxe graphique de l'ObCS

Bien que la structure d'un réseau de Petri soit donnée par ses fonctions d'incidences (*Pré* et *Post*), on préfère en général considérer sa représentation graphique, plus lisible. Les conventions graphiques suivantes, illustrées en Figure III.4 sont appliquées pour la description d'un ObCS (que ce soit un ObCS de spécification ou un ObCS d'implémentation, que nous décrivons ci-après) :

- Le nom d'une place apparaît en *italique* à l'intérieur de l'ellipse qui la représente. Pour alléger la représentation graphique du réseau, et pour minimiser les croisements d'arcs qui diminuent sa lisibilité, on s'autorise à représenter graphiquement plusieurs fois la même place. Toutes les ellipses où figure le même nom référencent donc la même place, et les arcs incidents à cette place sont l'union des arcs incidents à ses représentations graphiques⁷. Certaines places, qui décrivent des états ou des conditions transitoires, peuvent être représentées par des ellipses anonymes. Bien entendu chacune de ces ellipses anonymes dénote une place différente ;

⁷Cette facilité graphique doit être utilisée avec discernement car un usage abusif pourrait rendre la lecture de l'ObCS plus difficile, à l'encontre de l'effet escompté.

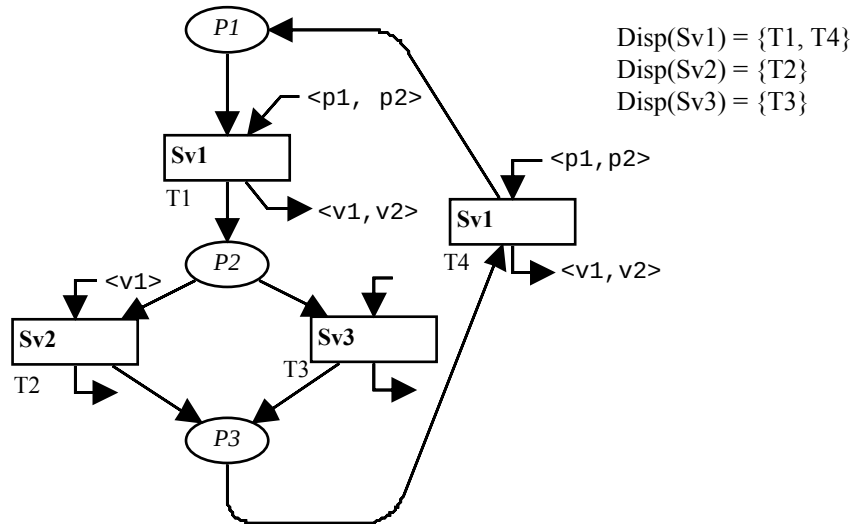


Figure III.4 - Syntaxe graphique de la pragmatique d'une classe

- Le nom d'une transition apparaît à côté du rectangle qui la représente. Dans de nombreux cas, on s'abstiendra de faire figurer le nom des transitions sur la représentation graphique des réseaux. Si la transition est associée à un service, le nom de ce service apparaît en **gras** à l'intérieur du rectangle qui la représente. On a souligné le fait que le même service peut être associé à plusieurs transitions de l'ObCS. On trouvera donc fréquemment dans un ObCS des rectangles portant le même nom de service ;
- Si une transition comprend une action, le texte de celle-ci apparaît en **caractères de listing** à l'intérieur du rectangle.

La Figure III.4 montre un ObCS correspondant à la définition de l'interface de la Figure III.3. La fonction de disponibilité, ajoutée en légende, est décrite graphiquement par les arcs d'activation et de complétion, et par la suite nous ne donnerons plus de définition textuelle des fonctions de disponibilité.

3.3. Instanciation et rapport Classe / Instance

- La spécification de chaque classe contient la définition d'une opération de création (notée **Create**), qui détermine de quelle manière sont créées les instances de cette classe. Cette opération n'est pas un service (elle ne peut pas être appliquée à une instance déjà créée) mais est notée comme une fonction qui prend une liste de paramètres d'entrée et a pour valeur de retour une liste à un seul élément, dont le type est la classe considérée, et qui dénote le nom de l'objet nouvellement créé. Un appel à l'opération Create de la classe C_b sera noté :

$$V_b := C_b.Create < \text{liste de paramètres} >$$

où V_b est une variable de la classe C_b ou d'une classe ancêtre dans la hiérarchie d'héritage (cf §IV.2).

Dans un LOO classique, une classe définit la structure de données de ses instances par un ensemble d'attributs. Toutes les instances ont donc la même structure, mais diffèrent par les valeurs de ces attributs.

Les Objets Coopératif respectent cette vision, et l'étendent au comportement des objets décrit par l'ObCS : tous les objets d'une classe ont le même comportement (ils partagent le même ObCS) mais possèdent un **marquage différent** du RPO de cet ObCS.

Nous verrons en §V.3 Etape 3 comment les attributs d'un objet sont en fait gérés de la même façon que le marquage de son ObCS. Le fait de mémoriser une information relative à un objet par un attribut ou par un élément de son marquage est un choix de modélisation, qui est sans incidence sur la structure profonde de cet objet : à tout instant, l'état d'un objet est complètement caractérisé par le marquage de son ObCS.

L'opération *Create* doit définir l'état initial de l'objet, c'est à dire la valeur initiale de ses attributs et le marquage initial de son ObCS.

L'initialisation des attributs est décrite en utilisant la syntaxe classique de l'affectation. Les marquages initiaux sont spécifiés en utilisant une syntaxe spéciale d'affectation, que nous décrivons par une BNF :

```

<Marquage d'une place> ::=
    <nom de place> := <multienemble de jetons>
<multienemble de jetons> ::=
    <multiplicité> • <jeton>
<multienemble de jetons> ::=
    <multiplicité> • <jeton> , <multienemble de jetons>
<multiplicité> ::=
    <INTEGER>
<jeton> ::=
    <>
<jeton> ::=
    < <liste de valeurs> >
<liste de valeurs> ::=
    <Valeur>
<liste de valeurs> ::=
    <Valeur> , <liste de valeurs>

```

Figure III.5 - BNF de la définition du marquage initial d'une place

Les Figures III.6 et III.7 illustrent l'utilisation de cette syntaxe d'affectation dans le code d'une opération *Create*. Les valeurs dans la définition d'un n-uplet peuvent être des paramètres de l'opération *Create*, ou des constantes ; on peut s'abstenir de mentionner une multiplicité égale à 1.

Pour favoriser la lisibilité de l'ObCS, on fera figurer sur sa représentation graphique la distribution de jetons du marquage résultant de l'exécution de *Create*.

3.4. Exemples

Pour illustrer la spécification d'une classe dans le formalisme des Objets Coopératifs, nous allons décrire deux classes d'objets. Les entités décrites par ces classes n'appartiennent pas vraiment au

domaine d'application auquel nous destinons notre formalisme, à savoir les systèmes concurrents. Au contraire nous avons choisi ces entités parmi les éléments de base de la culture informatique, afin que le lecteur puisse se concentrer sur les aspects syntaxiques du formalisme, sans avoir au préalable à intégrer la sémantique du système modélisé.

- La classe *Fichier* décrit un fichier tel que le manipule un système d'exploitation de type UNIX. Ce fichier est une simple suite de caractères, chaque caractère étant individuellement adressable par sa position par rapport au début du fichier ;
- La classe *Fichier_protégé* décrit un fichier de même type, mais qui définit en plus une politique de gestion pour ses accès en lecture et en écriture.

a. La classe Fichier

La spécification complète de la classe *Fichier* se trouve en Figure III.6. Cette spécification ne publie ni types ni constantes, et on y distingue quatre parties :

- La définition des services : chaque service est défini par un nom⁸, sa liste de paramètres formels d'entrée figurant entre <> et sa liste de paramètres formels de retour, suivant la syntaxe définie au §III.3.2. Les mots-clés du langage (directement empruntés au langage Eiffel) sont mis en **gras** ;
- L'opération de création (*Create*) dont le code apparaît après la définition de tous les services ;
- La définition des places de l'ObCS : pour chaque place (désignée par son nom) cette partie définit son type, c'est à dire la manière dont sont construits les n-uplets de valeurs que cette place peut recevoir ;
- Enfin la représentation graphique de l'ObCS, qui utilise les conventions citées plus haut.

La fonctionnalité décrite par l'ObCS de *Fichier* est très simple : un objet instance de la classe *Fichier* est associé à un fichier physique du système d'exploitation. Cet objet est créé dans l'état *Fermé* et ne peut alors accepter que le service *Ouvrir*, qui le place dans l'état *Ouvert*. Une fois ouvert, un fichier accepte les services *Ecrire*, *Lire*, *position* et *Aller_en*, un nombre quelconque de fois, dans n'importe quel ordre et en exclusion mutuelle. L'acceptation du service *Fermer* le remplacera dans l'état *Fermé*.

Le paragraphe ci dessus peut être considéré comme une description informelle du comportement d'une instance de *Fichier*, qui n'échappe pas à tous les problèmes qui découlent de l'utilisation du langage naturel. On voit immédiatement l'intérêt de disposer d'un formalisme qui permette d'exprimer le même comportement d'une manière concise, formelle et non ambiguë.

⁸[Parnas 73] a montré combien il est dangereux d'accorder trop d'importance à la sémantique des noms d'opérations, même lorsqu'on manipule des types bien connus. Une bonne pratique consiste donc à documenter le service par un commentaire qui a pour but de préciser la sémantique du service, en essayant de trouver un juste équilibre entre le laconisme et le verbiage. On doit notamment éviter d'introduire des indications sémantiques redondantes (ou pire encore contradictoires) avec l'information qui est portée par l'ObCS.

```

Class Specification Fichier;
Services
  Ouvrir;
    -- Ouvre le fichier.

  Fermer;
    -- Ferme le fichier.

  Aller_en <position : INTEGER>;
    -- Met à position l'indice du caractère courant

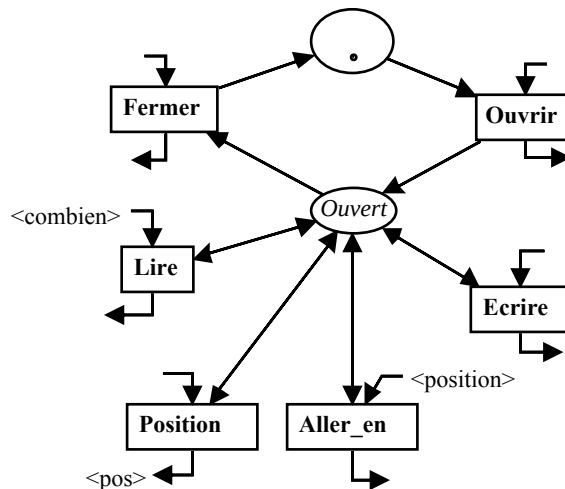
  Lire <combien : integer> : <chaîne : STRING>;
    -- Renvoie une chaîne d'au plus combien caractères lus
    -- à partir du caractère courant

  Ecrire <chaîne : STRING>;
    -- Ecrit chaîne à partir du caractère courant.

  position : <p : INTEGER>;
    -- Indice du caractère courant dans le fichier.

  Create <nom : STRING> is Fermé := <> end;
    -- l'objet est associé au fichier physique désigné par nom
    -- Initialement le fichier est fermé : la place Fermé est
    -- marquée par un jeton simple à la création de l'instance
obcs
  -- Définition du type des places
  Fermé, Ouvert : <> -- Les places Fermé et Ouvert reçoivent
    -- des jetons simples

```

Figure III.6 - Spécification de la classe *Fichier*

b. La classe *Fichier_protégé*

La classe *Fichier_protégé* décrit la solution que nous proposons pour le problème des lecteurs / rédacteurs, grand classique de la programmation concurrente s'il en est. Cette classe décrit un fichier du même type que celui modélisé par la classe *Fichier*, mais elle impose en plus une politique de gestion des accès en lecture et en écriture.

Les demandes d'ouverture du fichier sont maintenant qualifiées par leur type : un client ouvre le fichier en vue d'y effectuer soit des lectures, soit des écritures. Toute demande d'ouverture satisfaite

renvoie un identificateur de connexion, qui devra être ensuite passé en paramètre de tout autre accès au fichier. Cet identificateur de connexion est dénoté dans la spécification par une valeur de type *Connexion*, un type classe que l'on suppose défini par ailleurs.

Les autres services offerts par la classe sont identiques à ceux offerts par la classe *Fichier*, à l'exception du paramètre supplémentaire *conn* de type *Connexion*, qui dénote la connexion à laquelle l'invocation du service fait référence. Pour chaque connexion, la classe doit maintenir une position courante différente, et c'est à partir de cette position que s'effectuera la prochaine opération de lecture ou d'écriture pour la connexion concernée.

La politique de lecture / écriture est la suivante : on autorise plusieurs connexions en lecture simultanées. Une connexion en écriture interdit toute autre connexion (en lecture ou en écriture), et réserve le fichier à l'usage exclusif du rédacteur jusqu'à ce que celui-ci ferme sa connexion en invoquant le service *Fermer*.

La spécification de *Fichier_protégé* (figure III.7) fait apparaître des services associés à plusieurs transitions de l'ObCS. On voit notamment que les opérations *Fermer* et *Aller_en* sont chacune associées à deux transitions. L'indéterminisme introduit par cette construction sera levé dynamiquement à chaque invocation par unification sur la valeur du paramètre *conn*.

L'ObCS traduit fidèlement la politique décrite plus haut : on remarque notamment que la transition *Ouvrir_en_écriture* consomme le jeton présent dans la place *Pas_d'écrivain*, ce qui empêche tout déclenchement ultérieur des transitions *Ouvrir_en_lecture* et *Lire*. On remarque également que le service *Aller_en* pour un lecteur peut être exécuté concurremment avec d'autres services acceptés par l'objet, par exemple *Ecrire*.

Class Specification *Fichier_protégé*;**Services**

```

Ouvrir_en_lecture : <conn : Connexion>;
Ouvrir_en_écriture : <conn : Connexion>;
Fermer <conn : Connexion>;
Position <conn : Connexion> : <pos : INTEGER>;
Aller_en <conn : Connexion, position : INTEGER>;
Ecrire <conn : Connexion, quoi : STRING>;
Lire <conn : Connexion, combien : INTEGER> : <chaîne : STRING>;
Create <nom : STRING> is Pas_d'écrivain := <> end;

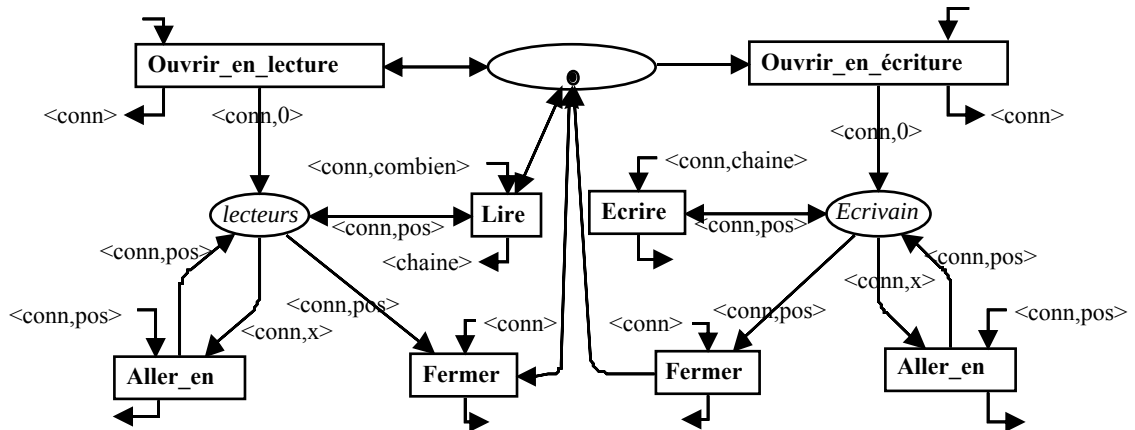
```

obcs

```

Pas_d'écrivain : <>;
Lecteurs, Ecrivain : <c : Connexion>;

```

Figure III.7 - Spécification de la classe *Fichier_protégé*

3.5. Interprétation de l'ObCS de spécification

L'activité d'un objet consiste à exécuter son ObCS : un service offert par l'objet s'exécute lorsque une des transitions associée à ce service est franchie.

L'ObCS d'une classe, qui définit le comportement de ses instances, est décrit par un RPO et par la fonction de disponibilité. Pour que la sémantique de cet ObCS soit claire et non ambiguë, il faut que soient clairement définies la franchissabilité de ses transitions et la règle de tir à adopter (déclenchement concurrent ou exclusif des transitions).

La franchissabilité et la règle de tir des transitions sont définies ci-dessous de manière intuitive ; la franchissabilité des transitions de l'ObCS de spécification est précisée dans sa définition formelle (§ II.1.2). La règle de franchissement parallèle, quant à elle sera précisée au §V.4.

a. Franchissabilité des transitions

Dans le cas d'un ObCS de spécification, la franchissabilité des transitions de service est définie différemment de celle des transitions privées.

- Une transition privée est franchissable en fonction du marquage de l'ObCS conformément à la définition de la franchissabilité dans les RPO (cf. Définition II.19) ;

- Une transition de service est franchissable si et seulement si : elle est franchissable au sens des RPO, et il existe une invocation en attente pour le service considéré. Les paramètres réels de cette invocation du service sont alors unifiés avec les variables d'entrée de l'arc d'invocation de la transition.

A un instant donné, il peut exister plusieurs invocations en attente pour le même service. Le choix parmi ces invocations est alors indéterministe : en particulier la préservation de l'ordre d'émission (Définition I.6) n'est pas garantie, et une invocation peut être acceptée concurremment avec un autre (concernant le même service ou un service différent) si les transitions de services associées sont concurremment franchissables (Définition II.22). Nous présentons en §VI.2 des extensions au formalisme de base de OC, qui visent justement à permettre de préciser l'ordre de prise en compte des invocations en attente, aussi bien en fonction de leur ordre d'arrivée qu'à partir de critères de priorité bien définis.

b. Règle de franchissement concurrent

Les transitions d'un ObCS de spécification (qu'il s'agisse de transitions de service ou de transitions privées) sont une description abstraite des opérations qui agissent sur l'état de l'objet. Au stade de la spécification, on ne peut pas préjuger de la manière dont seront concrétisées par l'implémentation ces opérations abstraites. Le cas le plus fréquent est qu'une transition de l'ObCS de spécification sera éclatée dans l'implémentation en plusieurs transitions, et que l'exécution du service ne sera alors plus atomique. On doit donc considérer que plusieurs transitions peuvent être tirées concurremment dans l'ObCS de spécification. Plus précisément :

- Toutes les transitions rendues concurremment franchissables par le marquage de l'ObCS de spécification (cf. Définition II.22) peuvent être franchies en parallèle. La règle de franchissement n'impose pas toutefois qu'elles le soient, et toutes les combinaisons de franchissements concurrents ou séquentiels sont licites, pour autant qu'elles restent compatibles avec la définition du franchissement dans les RPO (cf. Définition II.19) ;
- Si deux transitions associées au même service sont franchissables, elles ne peuvent être franchies concurremment que pour des invocations différentes.

c. Description de l'interaction état / traitements

On a déjà souligné l'interdépendance très forte qui existe dans tout système entre les données qui le composent et les traitements auquel il peut donner lieu (cf § I.1). C'est justement la reconnaissance de cette interaction qui a donné ses fondations au modèle objet, qui rassemble en une seule entité ces deux composantes d'un système, ainsi qu'aux RPO. Nous pouvons à présent expliciter comment le formalisme des Objets Coopératifs répond à un de nos objectifs de modélisation, à savoir exprimer avec une notation concise et non ambiguë l'interaction entre l'état de l'objet et l'activation de ses services :

- L'activabilité d'un service dépend de l'état de l'objet : un service est activable pour un objet si le marquage de l'ObCS de cet objet rend franchissable une des transitions associées à ce service ;

- L'exécution d'un service modifie l'état de l'objet : cette modification est précisément définie par l'évolution du marquage qui résulte du franchissement d'une des transitions associées au service.

d. Accès aux attributs publics

Une classe définit pour ses instances un ensemble d'attributs, ou variables d'instance, certains publics (introduits dès la spécification), d'autre privés (introduits en implémentation, que nous décrivons ci-après). Les attributs servent de **registres** à l'ObCS de la classe où ils sont définis. Ils peuvent donc être utilisés dans n'importe quelle transition de celui-ci.

Les registres ont été définis en §II.2.2 comme des extensions des RPO, un registre représentant en fait une place cachée du réseau. La question de l'initialisation de ces registres et de l'accès à leur valeur (en lecture ou en écriture) par d'autres objets se pose donc.

- Les Objets Coopératifs ne définissent pas, au contraire d'Eiffel, de politique d'initialisation par défaut des attributs (valeur NIL pour les références, 0 pour les entiers et les réels, etc...). Nous n'avons pas non plus retenu la solution de certains langages tels que Pascal, pour lesquels une variable non initialisée a une valeur indéfinie. La solution retenue, qui est conforme à l'identification que l'on donne entre registre et jeton-typé dans une place, est à notre connaissance unique : un attribut non initialisé **est inaccessible**, et tout service ou opération faisant accès à cet attribut est **impossible** jusqu'à ce que l'attribut reçoive une valeur ;

L'opération *create* aura fréquemment pour rôle de donner une valeur initiale aux attributs de l'instance. Il n'est toutefois pas toujours significatif de fournir les valeurs de tous les attributs au moment de la création. C'est même parfois impossible, notamment lorsqu'on souhaite avoir une relation d'utilisation cyclique entre instances (un exemple significatif est traité au § IV.3.2.) ;

- L'interface d'une classe est constituée de services, et d'attributs qui sont accessibles en lecture seulement. L'accès en écriture à des attributs publics d'une autre classe se fera donc, comme en Smalltalk [Goldberg...83], en fournissant des services conçus à cet effet. Pour alléger la structure de l'ObCS et pour faciliter la tâche du concepteur, deux méthodes d'accès à un attribut sont prédéfinies par le formalisme, qui précise leur syntaxe et leur sémantique :
 - Un **service d'initialisation** a pour but de fixer la valeur initiale d'un attribut. Par convention, un tel service aura pour nom **SETattribut** où *attribut* désigne le nom de l'attribut à initialiser. Un service d'initialisation a un seul paramètre en entrée, qui désigne la valeur d'initialisation de l'attribut et n'a pas de paramètre en sortie. Il ne peut être exécuté qu'une fois ;
 - Un **service de mise à jour** a pour but de changer la valeur d'un attribut. Un tel service a pour nom **CHANGEattribut**, a un seul paramètre en entrée, qui désigne la nouvelle valeur de l'attribut et a pour valeur de retour son ancienne valeur. Il ne peut être exécuté que si l'attribut a été initialisé, soit par l'opération *Create*, soit par un service d'initialisation.

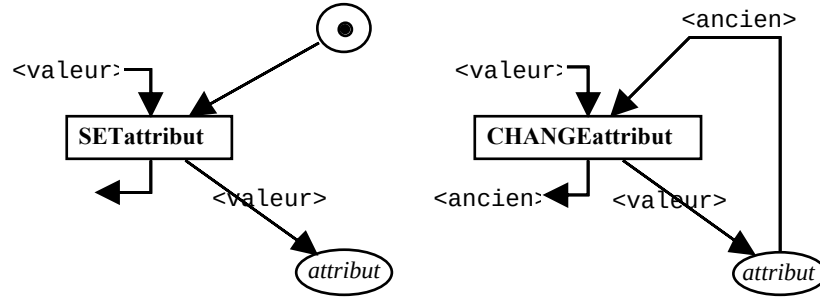


Figure III.8 - Services d'initialisation et de mise à jour

La structure de contrôle et la sémantique de ces services de manipulation d'attributs est illustrée en Figure III.8. Une classe ne publie pas forcément les services *SET* et *CHANGE* sur chacun de ses attributs publics. Pour qu'un tel service soit disponible, il faut qu'il soit déclaré dans l'interface de la classe. Le concepteur peut alors se dispenser de faire figurer les transitions associées à ce service dans l'ObCS ; la place / attribut étant «invisible» dans l'ObCS, la transition associée à un service d'initialisation ou de mise à jour semblerait non connexe au reste du réseau.

3.6. Définition formelle

Définition III.1 - Spécification d'une classe d'Objets Coopératifs

La **spécification** d'une classe d'Objets Coopératifs est un sept-uplet

$\text{Spec} = \langle R, \text{Att}, \text{Ty}, \text{Co}, \text{Sv}, \text{Sig}, \text{Disp}, \text{Mo} \rangle$ où :

- $R = \langle C, T, V, P, \text{Pré}, \text{Post} \rangle$ est un RPO, défini comme au chapitre II. R est muni d'un ensemble de **registres** (cf §II.2.2), Att ;
- On note $U = \bigcup_{t \in C} \text{Dom}(t)$ l'**univers** de la classe ;
- $\text{Ty} \subseteq C$ est l'ensemble des **types exportés** par la classe ;
- $\text{Co} \subseteq U$ est l'ensemble des **constantes exportées** par la classe ;
- $T = \text{TS} \cup \text{TP}$ et $\text{TS} \cap \text{TP} = \emptyset$, partition des transitions de l'ObCS en transitions de service (TS) et transitions privées (TP) ;
- Sv est l'ensemble des **services** de la classe ;
- $\text{Sig} : \text{Sv} \rightarrow (V \times C)^* \times (V \times C)^*$ est la fonction signature. Elle associe à chaque service une liste de couples (variable, type) définissant sa **signature d'entrée** et une liste de même structure définissant sa **signature de sortie** ;
- $\text{Disp} : \text{Sv} \rightarrow \mathcal{P}(T)$ est la **fonction de disponibilité**, telle que :
* $\forall s \in \text{Sv}, \text{Disp}(s) \neq \emptyset$

- * $\forall s, s' \in Sv, \text{Disp}(s) \cap \text{Disp}(s') = \emptyset$ si $s \neq s'$
- * $TS = \bigcup_{s \in Sv} \text{Disp}(s)$ est l'ensemble des **transitions de service** ;
- * $TP = T \setminus TS$ est l'ensemble des **transitions privées** ;

La fonction Disp matérialise l'association des services offerts par la classe à des transitions du RPO R . Le couple (R, Disp) sera appelé l'**ObCS de la classe** ;

- $Mo : P \rightarrow \mathbb{N}(U^*)$ est le **marquage initial** de chaque instance de la classe. Ce marquage est défini de manière textuelle par le code de l'opération *Create*.

Définition III.2 - Règle d'évolution d'un ObCS

Un objet change d'état en exécutant des services, ou en franchissant des transitions privées. A partir d'un marquage M de son ObCS, un objet peut exécuter un service $s \in Sv$ si et seulement si :

- Il existe une **requête en attente** sur ce service. La valeur des paramètres d'entrée de la requête sont unifiés avec les variables d'entrée de $\text{Sig}(s)$;
- Il existe $t \in \text{Disp}(s)$ tel que $M \xrightarrow{t}$.

On appelle **exécution du service** s le franchissement de la transition t .

3.7. Propriétés de l'ObCS de spécification

L'évolution d'un Objet Coopératif étant décrite par un RPO, nous pouvons profiter des techniques d'analyse statique dédiées aux réseaux de Petri, afin d'étudier les propriétés du comportement de chaque objet d'une classe.

Avant tout, il nous faut prendre en compte un problème lié à la nature même des ObCS : un sous système ouvert n'est pas destiné à évoluer seul, et son comportement ne peut être étudié sans émettre quelques hypothèses sur le comportement de son environnement. L'ObCS d'une classe modélise justement un tel système ouvert, pour lequel les transitions de service définissent l'interface avec l'environnement.

Avant de pouvoir appliquer les méthodes classiques d'analyse des RdP aux ObCS, il est nécessaire de **fermer** ces réseaux, de la manière suivante :

La **clôture** d'un ObCS est une opération qui consiste à supprimer les arcs d'activation et de complétion adjacents aux transitions de service du réseau. L'hypothèse sous jacente est que l'environnement du réseau fournit à tout instant les jetons nécessaires à son évolution, et consomme les jetons produits : l'environnement n'applique donc aucune contrainte au réseau. Il faut noter qu'avec cette approche, les **valeurs** des jetons reçus de l'environnement peuvent être définies par le réseau lui-même (par unification sur les arcs d'entrée des transitions de service, ou par préconditions), et dans le cas contraire ne peuvent être prises en compte.

Les méthodes classiques d'analyse des RdP peuvent s'appliquer au RdP sous jacent (§II.1.3) à la clôture d'un ObCS de spécification.

Il peut se révéler particulièrement intéressant d'interpréter les "bonnes" propriétés [Valette... 88] d'un réseau (borné, vivant, réinitialisable) en termes d'Objet.

- La propriété la plus importante à étudier sur un ObCS de spécification est la **vivacité** des transitions de service : si une transition de service est vivante, alors il existe toujours une solution permettant de répondre à une invocation concernant ce service, et l'appelé aura toujours le moyen de ne pas laisser l'appelant bloqué en attente de la réponse.

Il n'est toutefois pas nécessaire que toutes les transitions de service soient vivantes : on trouvera fréquemment des transitions qui ne peuvent être franchies qu'une fois, et qui correspondent par exemple à des initialisations obligatoires pour que l'objet puisse fonctionner (cf les services d'initialisation, §III.3.5). Il est toutefois nécessaire que toute transition de service soit quasi-vivante, sans quoi elle est inutile dans le réseau. On peut facilement constater que toutes les transitions des ObCS de spécification de *Fichier* et *Fichier_protégé* sont vivantes ;
- Le calcul de cycles dans les franchissements de transitions permet d'identifier les **tâches** ou les **procédures** qu'effectue l'objet. Le réseau est vivant si toute transition appartient à un cycle.

Un objet n'a pas forcément un ObCS cyclique : au contraire, un objet peut être créé pour exécuter une séquence d'actions déterminées, et ensuite disparaître. On dira qu'un objet est **mort** si son marquage est tel qu'aucune transition de son ObCS n'est franchissable (par exemple un marquage vide, si toute transition de l'ObCS a au moins un arc d'entrée). Une classe peut définir un service (nommé par convention **suicide**) tel que la transition de service associée vide complètement le marquage de son ObCS ;
- Le caractère **borné** des places des ObCS peut donner des indications précieuses quant à leur sémantique : on peut par exemple vérifier que la place *Ecrivain* de l'ObCS de *Fichier_protégé* est bornée à 1, établissant ainsi que le fichier tolère une seule ouverture en écriture à la fois. La place *Lecteurs*, au contraire, n'est pas bornée, car à partir du marquage initial la transition *Ouvrir_en_lecture* est indéfiniment franchissable : à ce niveau de description, en effet, on ne se soucie pas de limites physiques d'implémentation qui limiteraient forcément le nombre de lecteurs ;
- Le **calcul d'invariants** (cf §II.1.3) peut permettre d'établir des propriétés importantes de l'objet, telles que l'exclusion mutuelle entre services ou des contraintes de séquence (chapitre IX).

4. IMPLEMENTATION D'UNE CLASSE

Le fonctionnement d'un système est décrit dans le formalisme des Objets Coopératifs par la coopération entre les objets qui le composent, et qui entretiennent des relations de type client / serveur, en communiquant par invocation. La spécification d'une classe décrit comment les instances se comportent du point de vue de leurs clients. L'implémentation d'une classe va décrire son fonctionnement interne.

Comme dans un langage classique, l'implémentation d'une classe d'Objets Coopératifs décrit une structure de données, une structure de contrôle et des algorithmes. L'implémentation d'une classe doit également définir comment ses instances vont utiliser d'autres objets du système, soit pour remplir les services offerts par sa spécification, soit pour ses besoins propres.

4.1. Structure de données

La spécification d'une classe peut exporter une liste d'attributs, qui sont qualifiés d'**attributs publics** de la classe. Dans l'implémentation, on peut compléter cette liste par la définition d'**attributs privés**, accessibles seulement à l'intérieur de la classe. La structure de données d'une classe est alors l'ensemble de ses attributs publics et privés.

Les types exportés par la spécification d'une classe sont connus dans son implémentation, sans qu'il soit nécessaire de les répéter. Les types exportés avec la mention *private* doivent recevoir une définition explicite dans l'implémentation.

4.2. Opérations internes

L'implémentation d'une classe peut définir un ensemble d'opérations internes, exprimées dans un langage algorithmique quelconque (dans la suite on adoptera pour les opérations internes la syntaxe du langage Eiffel). Les opérations internes sont utilisées pour faire des calculs exclusivement **dans le contexte de l'objet**, et non pas pour effectuer des communications.

Ces opérations peuvent être procédurales ou fonctionnelles, avec une liste de paramètres d'entrée et un paramètre de sortie.

Les opérations internes peuvent faire usage dans leur code de leurs paramètres, de variables locales, ou des attributs définis dans la structure de donnée de la classe.

Le code d'une opération interne ne peut pas invoquer une variable de type-classe, ni accéder à ses attributs publics⁹. Cette restriction est très contraignante, et restreint beaucoup l'étendue de ce que l'on pourra exprimer dans les algorithmes des opérations internes ; elle est toutefois nécessaire pour que la sémantique du formalisme puisse être entièrement décrite en restant strictement dans le cadre des réseaux de Petri : il est nécessaire que toute communication entre objets soit explicite, et non pas "cachée" au sein d'algorithmes.

Toute communication entre objets (c'est à dire toute invocation ou tout accès à un attribut public) sera donc spécifiée en tant qu'action d'une transition de l'ObCS d'implémentation.

⁹ Ces attributs peuvent toutefois lui être communiqués en tant que paramètres de l'appel.

Nous présentons toutefois au chapitre VI une extension au formalisme des Objets Coopératifs qui permet de décrire aisément des objets dont la complexité est essentiellement algorithmique, en s'affranchissant de la contrainte citée ci-dessus : il s'agit des **objets passifs** qui sont comparables aux objets ayant un comportement séquentiel que l'on peut décrire par des langages tels que C++, Eiffel, etc.

4.3. Structure de contrôle : ObCS d'implémentation

La structure de contrôle d'une classe est décrite par un **ObCS d'implémentation** défini, comme l'ObCS de spécification, par un Réseau de Petri à Objets et la fonction de disponibilité. Ce réseau a pour but de décrire trois aspects de la structure de contrôle, qui sont distincts mais fortement interdépendants :

- La structure de contrôle décrit le **comportement** d'un objet de la classe, c'est à dire la manière dont il évolue, soit de manière spontanée, soit en réponse à des invocations de service ; l'exécution d'un service par l'objet se traduit par des franchissements de transitions dans son ObCS, et donc par une évolution de son état ;
- Elle spécifie ensuite les communications entre objets et par là même la **coopération** qui est nécessaire entre l'objet décrit et d'autres objets du système ;
- Elle montre enfin comment est structuré le **code algorithmique** qui doit être exécuté lors de l'appel d'un service.

Comportement et coopération

L'activité d'un Objet Coopératif peut avoir trois origines :

- Il répond à des invocations, agissant alors en tant que serveur ;
- Il invoque d'autres objets, se plaçant alors en position de client ;
- Il exécute des opérations internes, agissant sous son propre contrôle.

L'ObCS d'implémentation doit décrire ces trois aspects. Il est structuré comme l'ObCS de spécification, mais est plus détaillé et contient des informations qui ne sont pas pertinentes pour les clients :

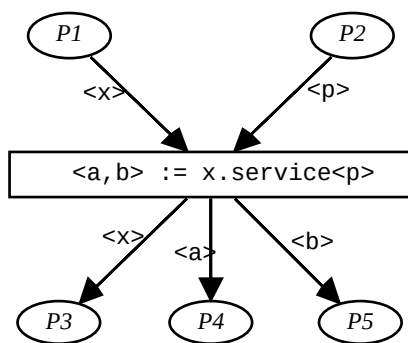
- **Réponse à des invocations** : Comme pour l'implémentation, l'ObCS d'implémentation définit la disponibilité des services ; il décrit en plus très précisément le traitement associée à ce service : chaque service défini dans l'interface de la classe est associé non plus à une ou plusieurs transitions, mais à des **couples de transitions**, pour lesquels :
 - * une des transitions comporte en entrée un arc d'activation étiqueté par les paramètres d'entrée du service ; cette transition est appelée **transition d'accord** du service, et définit sous quelle condition une invocation relative à ce service peut être acceptée par l'objet ;

- * l'autre comporte en sortie un arc de complétion étiqueté par les paramètres de sortie ; cette transition est appelée **transition de retour** du service, et définit quand et comment le client sera informé de l'accomplissement du service, et quel résultat il obtiendra.

Une transition d'accord contient le nom du service auquel elle est associée, et une transition de retour contient ce même nom, préfixé par le mot-clé **End_**. La transition d'accord et la transition de retour peuvent éventuellement être confondues, et dans ce cas la transition contiendra seulement le nom du service. Il peut exister plusieurs transitions d'accord pour un même service, spécifiant ainsi qu'un service peut avoir plusieurs conditions de déclenchement différentes. Il existe toutefois une et une seule transition de retour par service ;

- **Invocations d'autres objets** : Le premier moyen de communication entre objets est l'**invocation** (cf §I.1). Cette communication est spécifiée dans l'ObCS d'implémentation par des transitions dont l'action est un appel de services et qui sont appelées des **Transitions d'Invocation (TI)**. La syntaxe d'une invocation est, classiquement, la notation pointée :

*<tuple de variables> :=
 Serveur.service_invoqué <paramètres de l'appel>*



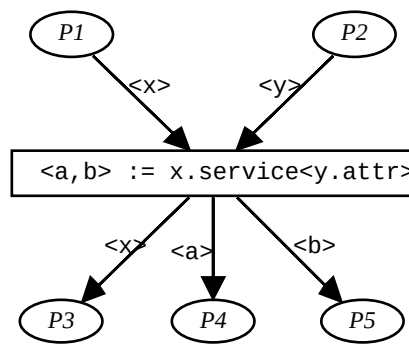
ObCS :

P1, P3 : <x : Serveur>

P2, P4, P5 : <i : INTEGER>

Figure III.9 - Exemple de transition d'invocation

L'objet serveur peut être dénoté soit par une variable d'entrée de la TI, soit par un attribut-référence de la classe cliente. Les paramètres de sortie du service sont affectés à des variables de sortie de la transition. L'invocation est la matérialisation d'un **flot de contrôle** entre l'objet client et l'objet serveur. La figure III.9 montre une TI où l'objet serveur et le paramètre d'entrée transmis sont dénotés par des variables d'entrée de la transition ; le service invoqué renvoie un couple de valeurs, dont la figure illustre l'affectation à des variables de sortie de la transition.



ObCS :

P1 : <x : Serveur1>

P2 : <y : Serveur2>

P4, P5 : <i : INTEGER>

...

Class Specification Serveur2

...

Attributes

attr : INTEGER;

...

Figure III.10 - Transition d'invocation et d'accès

Le deuxième moyen de communication entre objets est l'accès par un objet client à un attribut publié par un objet serveur ; on rappelle qu'un attribut public est accessible en lecture seulement. La syntaxe en est elle aussi classique :

serveur.attribut_public

Une transition qui contient une telle expression sera appelée une **Transition d'Accès (TA)**. Il faut remarquer qu'une transition peut être à la fois TI et TA, comme le montre la Figure III.10. L'accès à un attribut ne modifie pas l'objet serveur : un tel accès matérialise donc simplement un **flot de données** entre serveur et client.

- **Opérations internes** : Les transitions de l'ObCS d'implémentation qui ne sont pas des TI peuvent comporter une action qui est définie comme : - Un appel d'opération interne procédurale, de la forme

opération(p₁, p₂, ..., p_n) ;

- Une affectation de la forme

var := expression ;

où *var* est une variable de sortie de la transition ou un attribut de la classe, et où *expression* peut contenir une combinaison arbitraire d'appels d'opérations internes fonctionnelles.

Le sous-réseau compris entre une transition d'accord et la transition de retour d'un service est appelé la Structure de Contrôle de l'Opération ou **OpCS** (Operation Control Structure) par référence à la même notion dans la méthode HOOD. L'OpCS montre quels peuvent être les enchaînement d'actions effectuées lors d'une invocation concernant ce service. Un OpCS peut être purement séquentiel, ou comporter du parallélisme décrit par la structure du sous-réseau.

Un service pouvant avoir plusieurs transitions d'accord, il peut y avoir plusieurs OpCS pour le même service ; ceci permet de spécifier que le traitement d'une opération dépend de l'état de l'objet serveur.

Certains critères structurels doivent être respectés par les OpCS, visant notamment à assurer qu'un service accepté pourra toujours être mené à son terme. Ces critères sont détaillées au chapitre IX.

4.4. Exemples

Nous allons présenter l'implémentation de classes d'objets au sein du formalisme des Objets Coopératifs en utilisant les mêmes exemples qu'au III.3.4 : les classes *Fichier* et *Fichier_protégé*.

a. Classe Fichier : Implémentation

L'implémentation de la classe *Fichier* (Figure III.11) va faire de cette classe un serveur pur, réalisée sans utiliser les services d'aucune autre classe. On définit dans l'implémentation des opérations internes, qui se résument à des appels aux système d'exploitation sous-jacent. Ces appels sont exprimés en utilisant des primitives définies par le standard ANSI du langage C et nous utilisons pour cela la syntaxe définie par le langage Eiffel pour l'appel de routines écrites dans un autre langage. Pour alléger l'implémentation de la classe, on s'est dispensé de traiter les cas d'erreur qui pourraient être levés par l'appel aux primitives C. Une implémentation complète devrait bien entendu définir une politique de gestion des erreurs.

Pour cette classe, L'ObCS d'implémentation reste identique à l'ObCS de spécification, intégrant simplement les appels aux opérations internes en tant qu'actions des transitions.

```

Class Implementation Fichier
Types
  PtFile : ...    -- Représentation adéquate pour le type C : File *
Attributes
  chemin : STRING;
  tampon : STRING;
  stream : PtFile;
Operations
  FPosition : INTEGER is
    -- Indice du caractère courant dans le fichier.
    external fpos language "C"
    do
      result := fpos(stream)
    end;    -- fposition

  FOuvrir is
    -- Ouvre le fichier.
    external fopen language "C"
    do
      stream := fopen(chemin, "r+");
    end; -- FOuvrir

  FFermer is
    -- Ferme le fichier.
    external fclose language "C"
    do
      fclose(stream);
    end;    -- FFermer;

  FAller_en (position : INTEGER) is
    -- Met à position l'indice du caractère courant
    external fseek language "C"
    do
      fseek(stream, position, 0)
    end;    -- FAller_en

  FLire (combien : INTEGER) : STRING is
    -- Renvoie une chaîne d'au plus combien caractères lus
    -- à partir du caractère courant
    external fread language "C"
    do
      FAller_en(FPosition(stream));

```

```

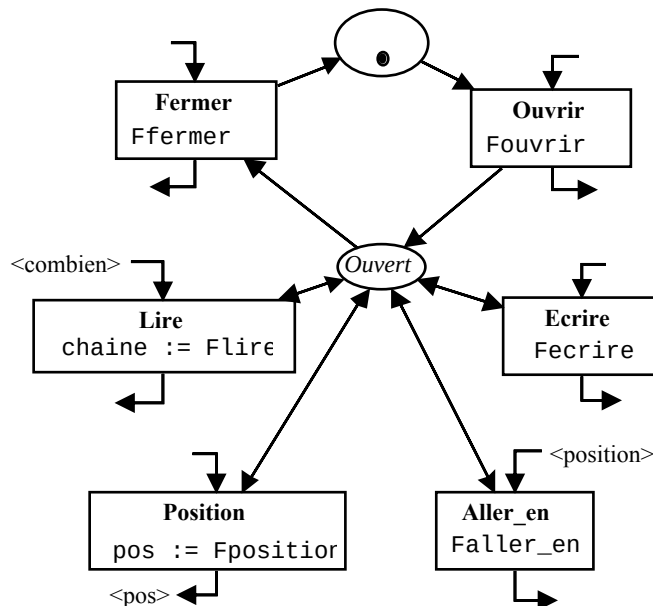
        tampon.retailler(combien);
        fread(tampon.en_c, 1, combien, stream);
        result := tampon;
    end;
    -- FLire

    FEcrire (chaîne : STRING) is
        -- Ecrit chaîne à partir du caractère courant.
    external fwrite      language "C"
    do
        fAller_en(fPosition(stream));
        fwrite(chaîne, 1, chaîne.longueur, stream);
    end;
    -- FEcrire

    create <nom : STRING> is
        -- Initialisation des attributs
    do
        chemin := nom;

        -- Définition du marquage initial de l'ObCS
        Fermé := <>;
    end;
    -- create
obcs

```

Figure III.11 - Implémentation de la classe *Fichier*

b. Classe *Fichier_protégé* : implémentation

L'implémentation de la classe *Fichier_protégé* (Figure III.12) va nous permettre d'illustrer la relation d'utilisation entre classes d'objets. Une instance de *Fichier_protégé* va utiliser une instance de *Fichier* pour réaliser les services qui lui sont demandés. L'instance de *Fichier* manipulée est représentée par un élément du marquage de l'ObCS de *Fichier_protégé*.

On remarque tout d'abord que le passage de la spécification à l'implémentation a essentiellement consisté à éclater les Transitions de Service (TS) de l'ObCS de spécification en un sous-réseau plus complexe. Les zones grisées sur la figure matérialisent ces «macro-transitions».

Le type des places à lui aussi évolué : ainsi la place *Pas_d'écrivain* qui était du type jeton simple (<>) dans la spécification est du type <Fichier> dans l'implémentation : en effet une instance de *Fichier_protégé* a besoin d'un serveur de classe *Fichier* pour exécuter ses services ; l'utilisation du *Fichier* est spécifiée dans les Transitions d'Invocation par des requêtes telles que *f.ouvrir*, *f.fermer*, *f.position*...

```
Class Implementation Fichier_protégé
attributes
  c_l : INTEGER;-- Combien de Lecteurs sont actuellement connectés
  Create <nom : STRING> is
    c_l := 0;
    Pas_d'écrivain := Fichier.Create(nom);
  end -- Create
obcs
  Pas_d'écrivain : <f : Fichier>;
  Ecrivain : <conn : Connexion, pos : INTEGER, f : Fichier>
  Lecteurs : <conn : Connexion, pos : INTEGER>
```

Figure III.12 - Implémentation de la classe *Fichier_protégé*

L'implémentation introduit un nouvel attribut, *c_l*, qui mémorise le nombre de lecteurs connectés ; cet attribut est utile pour déterminer à quel moment le *Fichier* utilisé doit être effectivement ouvert ou

fermé : on n'ouvre le fichier que pour la première connexion, et on ne le ferme qu'à la déconnexion du dernier lecteur, s'il n'y a pas d'écrivain.

L'enrichissement de la structure de l'ObCS nécessite la mise à jour de l'opération *Create* : cette opération initialise maintenant l'attribut *c_l* à zéro, et crée l'instance de *Fichier* qui sera utilisée dans les TI.

On peut examiner de plus près les OpCS de quelques opérations, qui sont caractéristiques des structures de contrôle internes associées aux services.

L'opération *Ouvrir_en_écriture* effectue un test (décrit par une règle d'émission) sur l'attribut *c_l* afin de déterminer si le *Fichier* doit être ouvert. Elle génère un nouvel identifieur de connexion (forcément unique, car il s'agit d'un nouveau nom d'objet) qui est renvoyé au client. L'OpCS décrit précisément l'**effet de bord** de l'exécution du service sur l'objet serveur : le jeton de classe *Fichier* manipulé passe de la place *Pas_d'écrivain* à la place *Ecrivain*.

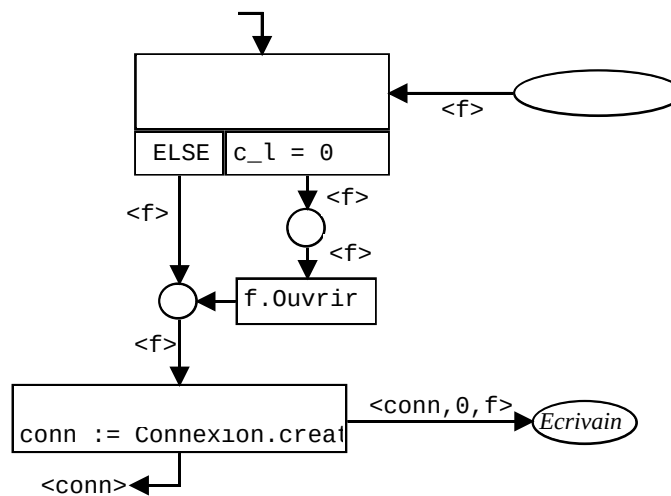


Figure III.13 - OpCS de l'opération *Ouvrir_en_écriture*

L'OpCS de l'opération *Lire* présente une caractéristique intéressante : le résultat de la lecture est renvoyé au client (par la transition *END_Lire*) avant que le traitement de l'opération soit complètement terminé pour le serveur, qui doit encore calculer la nouvelle *position* du *Fichier*.

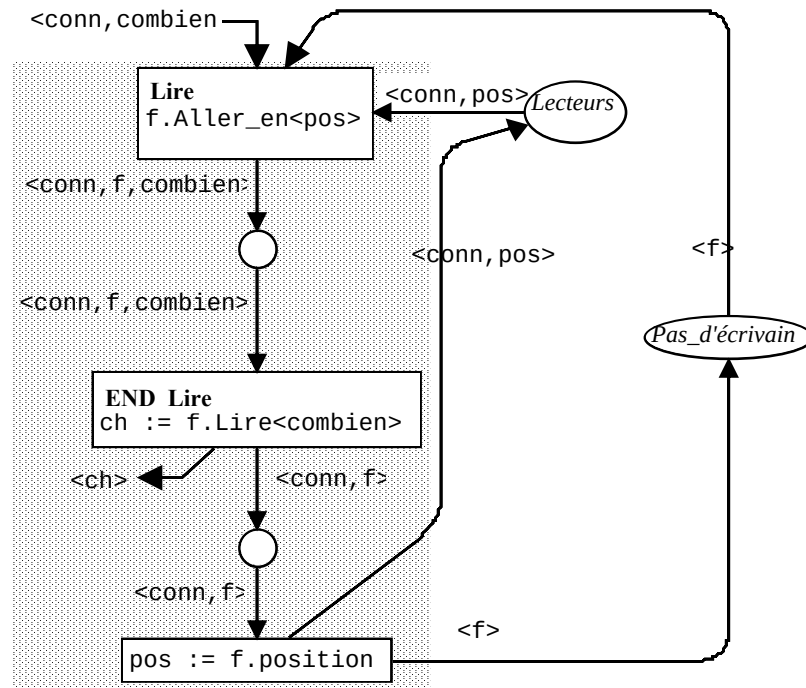


Figure III.14 - OpCS de l'opération Lire

L'exemple ci-dessus illustre une des caractéristiques de notre formalisme : les jetons qui circulent dans un ObCS sont fréquemment des **références d'objet**, qui en quelque sorte désignent un autre ObCS du système.

Dans notre exemple, l'exécution de l'opération *Ouvrir_en_écriture* va provoquer au moins deux changements d'état dans le système : elle va (s'il n'y a pas de lecteurs) faire passer l'objet *Fichier* de l'état *Fermé* à l'état *Ouvert* (cf l'ObCS d'implémentation de *Fichier*, Figure III.11), et va faire passer le jeton qui dénote l'objet *Fichier* de la place *Pas_d'Ecrivain* à la place *Ecrivain* dans l'ObCS de *Fichier_protégé* (on ne considère pas ici le changement d'état que va subir le client du service *Ouvrir_en_écriture*).

L'objet *Fichier*, après exécution de ce service, est dans l'état *Ecrivain* **relativement au *Fichier_protégé* qui le manipule**. Cette information n'est nulle part visible dans l'ObCS de *Fichier*, et n'a d'ailleurs pas de sens pour lui.

4.5. Interprétation de l'ObCS d'implémentation

On a explicité au paragraphe III.3.5 la manière dont l'ObCS de spécification doit être lu, c'est à dire la manière dont s'interprète le comportement d'un serveur. Pour que la vision de l'interaction entre objets soit complète, nous allons maintenant définir, également de manière intuitive, comment doit s'interpréter un ObCS d'implémentation. Il nous faut ici, non seulement préciser la définition de la franchissabilité et la règle de tir des transitions, mais aussi donner la signification d'une transition d'invocation (TI), dont l'action contient une requête de service. Comme pour l'ObCS de spécification, ces définitions sont données ici de manière intuitive, et seront formalisées au chapitre V.3.

a. Franchissabilité des transitions

On peut classer les transitions d'un ObCS d'implémentation en quatre catégories (non exclusives entre elles) :

- Les transitions d'accord, associées à un service par la fonction de disponibilité.
- Les Transitions d'Invocation (TI), dont l'action est une requête de service.
- Les Transitions d'Accès (TA), dont l'action est l'accès en lecture à un attribut public d'un autre objet.
- Les Transitions Privées (TP) : toutes les autres, y compris les transitions de retour (préfixées par END_) qui dénotent l'achèvement du traitement associé à un service.

La franchissabilité des transitions d'un ObCS d'implémentation se définit comme pour l'ObCS de spécification. Les transitions d'accord de l'ObCS d'implémentation ont la même règle de franchissement que les transitions de service de l'ObCS de spécification, c'est à dire que leur franchissabilité est conditionnée à la présence d'une requête en attente sur ce service. Une TA n'est franchissable que si l'attribut considéré a été précédemment initialisé.

b. Règle de franchissement concurrent

La règle de franchissement des transitions de l'ObCS d'implémentation peut être précisée : un objet effectue une seule opération à la fois, mais n'est pas bloqué dans son évolution pendant l'exécution des services qu'il réclame à d'autres objets. On doit donc différencier dans l'ObCS d'implémentation les TI des TA ou des transitions qui ne font appel qu'à des opérations internes de la classe.

- Les transitions dont l'action ne contient pas d'invocation de service sont franchies en exclusion mutuelle ;
- Les TI peuvent être tirées concurremment entre elles ou avec des transitions privées (TP).

Cette règle est justifiée par la sémantique d'une transition dont l'action contient une demande de service. Nous allons pour cela préciser les modalités de la communication entre l'objet client et l'objet serveur dans une requête de service.

4.6. Communication par rendez-vous

La Figure III.15 illustre une configuration typique de la communication entre un objet client et un objet serveur, en montrant des extraits de leur ObCS respectifs. Comme nous l'avons précisé au § III.4, toute communication entre objets est définie dans l'ObCS du client par des TI, transitions dont l'action est une invocation, et dans l'ObCS du serveur par un couple de transitions (transition d'accord, transition de retour) modélisant l'exécution de ce service.

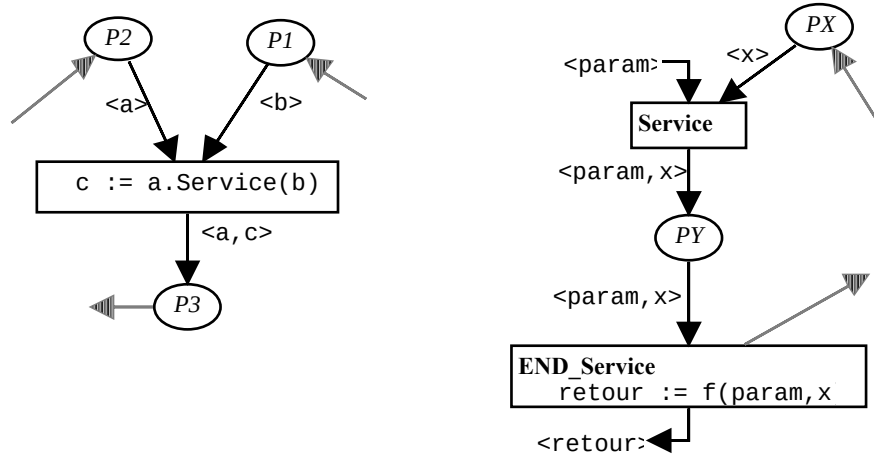


Figure III.15 - Les deux côtés d'une invocation

La sémantique intuitive que nous donnons à cette construction est proche du **rendez-vous** du langage Ada : toute communication est une synchronisation entre deux flots de contrôle. Le flot de contrôle appelant est bloqué jusqu'à ce que l'objet appelé accepte la requête et en termine l'exécution. Les deux flots de contrôle reprennent alors leur exécution indépendante.

Le RPO de la Figure III.16 explicite cette sémantique, en précisant la fonction des places et transitions qui interviennent dans l'accomplissement de la communication : côté client, la TI est éclatée en une **transition d'appel** et une **transition de complétion**, connectées par une place d'attente. Côté serveur, une **place paramètre**, qui contiendra le tuple de paramètres de l'appel est introduite en entrée de la **transition d'accord**, et de même une **place résultat** est introduite en sortie de la **transition de retour**.

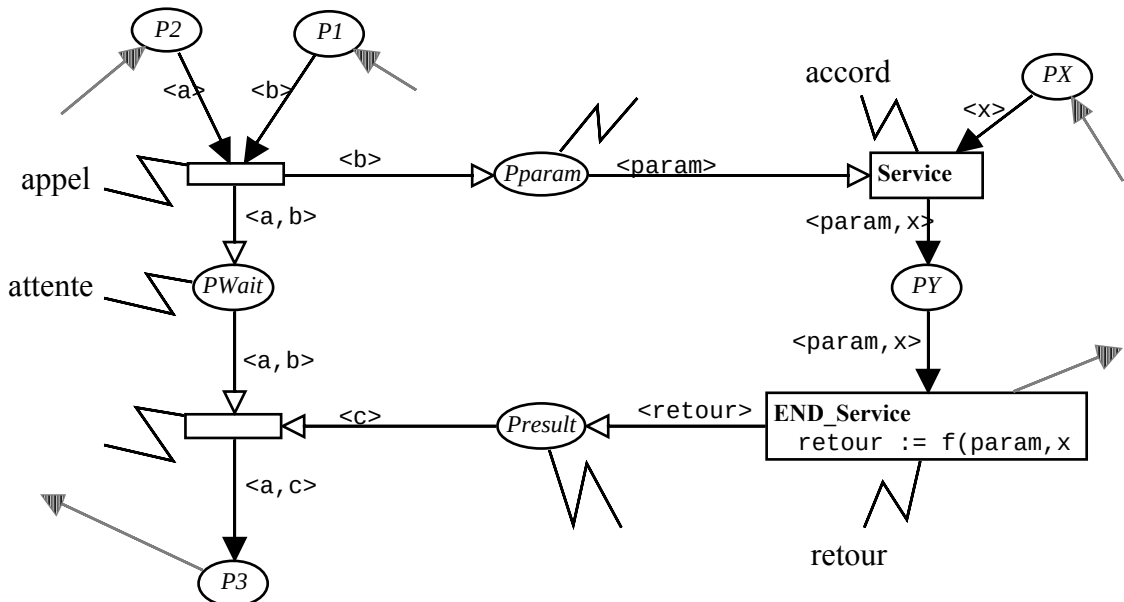


Figure III.16 - Sémantique intuitive d'une invocation

Il faut bien préciser que le réseau de la Figure III.16 ne sert qu'à donner une vision intuitive du traitement associé à une invocation de service. Ce réseau ne prend notamment pas en compte les problèmes liés à la dynamique des appels (une TI ne concerne pas toujours le même serveur), à l'auto-concurrence éventuelle des invocations de services et à la liaison tardive. Une définition formelle de la communication entre objets, traitant ces problèmes et compatible avec cette vision intuitive est donnée en § IX.

Cette construction modélise une **communication synchrone** entre objets : pour qu'une invocation s'exécute complètement, il faut qu'un flot de contrôle chez le client se synchronise avec un flot de contrôle chez le serveur. D'autres modalités possibles de communication sont décrites au chapitre VI.

4.7. Définition de la relation d'utilisation

On peut maintenant définir formellement la relation d'utilisation entre deux classes d'objet :

Définition III.3 - Relation d'utilisation

Une classe d'objets C_a **utilise** une classe d'objets C_b si une des transitions de l'ObCS d'implémentation de C_a contient une expression de la forme :

$V_b .Service < \text{liste de paramètres} >$

ou : $V_b .Attribut$

ou : $V_b := C_b .Create < \text{liste de paramètres} >$

où V_b est une variable de classe C_b .

Aucune restriction n'est imposée a priori quant à la structure du graphe de la relation d'utilisation. Ce graphe peut notamment comporter des cycles, et une classe peut être en relation d'utilisation avec elle-même, ce qui est le cas par exemple quand on décrit une structure de donnée récursive telle qu'un arbre. Certains domaines d'application pourraient imposer à ce graphe d'être un arbre, ou un treillis (logiciel structuré en couches, par exemple).

Lorsqu'on construit le graphe de la relation d'utilisation pour un ensemble de classes, on peut distinguer trois types de nœuds :

- Les nœuds standards, qui ont à la fois des antécédents et des suivants dans le graphe. Ce sont des classes qui jouent à la fois le rôle de client et de serveur ;
- Les nœuds racines, qui n'ont pas d'antécédents : ce sont des **clients purs**, dont aucun service n'est réclamé par une autre classe ;
- Les nœuds feuilles, qui n'ont pas de successeurs : ce sont des **serveurs purs**, qui ne réclament les services d'aucune autre classe. La classe *Fichier*, décrite au §III.3.4.a, est un serveur pur.

5. COHERENCE ENTRE SPECIFICATION ET IMPLEMENTATION

La cohérence syntaxique entre spécification et implémentation de la structure d'une classe est facile à assurer : il suffit que le nom et le nombre des services n'évolue pas, que la signature de ces services reste constante et que les attributs publics de la spécification soient conservés dans l'implémentation. Ces contraintes sont assurées par le formalisme en localisant ces informations dans la spécification et en ne les dupliquant pas dans l'implémentation.

La cohérence de la dynamique entre les deux descriptions d'une classe est plus difficile à assurer. Il faut que l'implémentation que l'on donne reste conforme à la spécification, et que l'objet que l'on implémente se comporte comme souhaité par le spécificateur.

Dans le formalisme des objets coopératifs, le comportement est décrit, aussi bien en spécification qu'en implémentation, par un Réseau de Petri. Nous pouvons dès lors caractériser de manière formelle la cohérence entre spécification et implémentation en utilisant les techniques d'analyse statiques développées pour les RdP.

Le principe que nous avons retenu est l'analyse et la comparaison des langages acceptés par les ObCS d'implémentation et de spécification. Nous avons défini un critère formel sur ces langages, qui permet d'assurer que l'on a correctement "implémenté la spécification". Ce point est repris et détaillé au chapitre IX, où nous avons regroupé l'étude des possibilités d'analyse offertes par les Objets Coopératifs.

6. POUVOIR D'EXPRESSION DU FORMALISME

L'utilisation du formalisme des RPO pour définir la pragmatique des objets d'une classe a des conséquences importantes quant à la typologie des objets que l'on peut décrire, et va nous permettre d'exprimer des comportements et des constructions qui dépassent le cadre des langages à objets traditionnels :

- L'utilisation principale des RdP est la modélisation de systèmes qui évoluent de manière discrète, mais pas forcément séquentielle ; dans un système modélisé par objets, l'évolution est provoquée par l'invocation par des objets de services qu'offrent d'autres objets. Ces invocations créent alors des flots de contrôle entre objets. Un système d'objets n'est pas séquentiel si plusieurs flots de contrôle peuvent évoluer concurremment, soit à l'intérieur d'un objet, soit entre plusieurs objets. En utilisant les RdP pour décrire la structure de contrôle des objets, on dispose de toute leur expressivité pour la spécification de flots de contrôle concurrents ;
- Si on considère que les transitions de l'ObCS de spécification peuvent être franchies concurremment, chacun des flots de contrôle qui traverse un objet peut évoluer en parallèle avec les autres. On peut donc aisément modéliser une concurrence interne à l'objet. Il faut remarquer que, d'après la règle de franchissement décrite plus haut (§III.4.5.b), les flots de

contrôle internes à un objet n'évoluent pas **simultanément** (les transitions privées sont franchies en exclusion mutuelle) mais **concurrentement**. La granularité d'exécution d'un objet est le franchissement d'une transition privée de son ObCS ;

- En associant plusieurs transitions au même service, on peut spécifier qu'une requête a des conditions d'activation et des effets différents suivant l'état de l'objet serveur ;
- On peut facilement au sein du formalisme décrire un **objet actif**, c'est à dire un objet qui peut provoquer un changement de son propre état sans pour cela être sollicité par un flot de contrôle en provenance de son environnement, et qui a la possibilité de générer de nouveaux flots de contrôle de sa propre initiative. Un objet coopératif peut donc avoir une **activité spontanée**, qui n'est pas déclenchée par l'activation d'un de ses services. Cette activité spontanée, qui peut représenter la "tâche de fond" de l'objet, est modélisée par des **transitions privées** de l'ObCS, c'est à dire des transitions qui ne sont pas associées à une invocation de service ;
- Les réseaux de Petri permettent de décrire des systèmes dont l'évolution n'est pas déterministe. La spécification d'un comportement non déterministe peut signifier que le système décrit est lui-même essentiellement non déterministe, ou que le niveau d'abstraction auquel on se trouve ne permet pas encore de définir précisément comment le choix est fait entre plusieurs alternatives d'évolution, ou encore que l'indéterminisme sera levé, au moment de l'exécution, par un mécanisme de commande extérieur au système (par exemple une interprétation au sens de [Valette 86]).

7. COMPARAISON AVEC L'APPROCHE AXIOMATIQUE

On peut comparer notre approche, qui consiste à définir la pragmatique d'une classe d'Objets (c'est à dire à la fois sa sémantique et sa dynamique) par un RPO, avec l'approche axiomatique classique promue par le langage Eiffel et héritée des Types Abstraits de Données. Dans cette approche la sémantique d'une classe d'objets est définie par un invariant portant sur la classe, et par des pré et post-conditions portant sur l'exécution des opérateurs

B. Meyer [Meyer 90a] a montré comment le modèle de la programmation par contrat peut être adapté aux systèmes concurrents ; nous avons présenté cette discussion en §I.1.3. La solution proposée est celle du contrat concurrent, qui interprète les préconditions comme des **conditions d'attente** à l'exécution d'un service. C'est bien l'interprétation que nous avons retenue dans notre formalisme, mais les préconditions, au lieu d'être décrites par des prédicats portant sur les attributs de l'objet, sont décrites par la structure même de son ObCS. Une requête concernant un service pour lequel aucune Transition de Service (TS) associée n'est franchissable correspond à une précondition fausse, et l'invocation correspondante est mise en attente jusqu'à ce qu'une des TS devienne franchissable.

Une clarification de vocabulaire s'impose ici. Il ne faut pas confondre les préconditions de Eiffel ou des TAD avec les préconditions que l'on associe aux transitions des RPO.

- Dans Eiffel, une précondition est une expression booléenne portant sur paramètres d'entrée du service et des attributs de l'objet serveur, qui définit les conditions pour qu'une invocation soit correcte, ou légale. Une invocation avec une précondition non vérifiée est une erreur et ne devrait jamais se produire dans un système «correct». Si une telle erreur se produit effectivement, ce ne peut être que par suite d'une erreur de programmation, qui doit provoquer l'arrêt du programme ;
- La signification d'une précondition d'une transition d'un RPO est toute autre : elle sert uniquement à choisir les entités avec lesquelles une transition peut être franchie, à lever des conflits effectifs (§II.2.1) entre transitions.

On pourrait toutefois envisager de faire jouer aux préconditions d'un RPO le même rôle que les préconditions de Eiffel, dans un cas bien précis : le cas d'une précondition sur une transition d'accord et qui porte uniquement sur les paramètres d'entrée du service :

- Avec la sémantique normale des préconditions dans les RPO, si une telle précondition n'est pas vérifiée, le franchissement de la transition d'accord ne pourra jamais avoir lieu, et le client restera pour toujours en attente de l'exécution de sa requête ;
- En donnant à cette construction la sémantique des préconditions Eiffel, un tel cas déclencherait une erreur d'exécution dépendante de l'implémentation physique (par exemple l'envoi d'un message d'erreur à un opérateur). Cette possibilité a déjà été évoquée par [De Bondelli 87] qui parle de **préconditions absolues** ou de **préconditions d'attente**.

Il existe un autre cas de blocage infini d'un client en attente de l'exécution d'un service : c'est le cas où un client invoque un serveur **mort**, c'est à dire un serveur pour lequel l'ObCS à un marquage qui ne rend franchissable aucune transition : Une invocation à un tel client ne pourra jamais être servie. Ce cas correspond également à une erreur de modélisation, et devrait lever une erreur d'exécution.

La définition d'une pragmatique par Réseaux de Petri présente d'autres différences importantes par rapport à une approche par TAD :

- Notre approche vise à fournir une description explicite et détaillée de l'espace d'états des objets, en précisant au maximum l'influence de l'état sur la disponibilité des services et réciproquement l'influence de l'exécution des services sur l'état de l'objet. Un TAD, au contraire, n'a pas d'état (plus précisément son espace d'états est défini de manière indirecte par les équations qui le régissent) et la notion d'«effet de bord» n'existe pas.
- Dans le langage Eiffel (que B.Meyer présente lui-même comme une implémentation possible des TAD), il est possible d'exprimer des contraintes relatives aux séquencements possibles entre les invocations, mais ceci nécessite souvent l'introduction de variables d'instance à cet effet : une implémentation en Eiffel de la classe Fichier décrite en §III.3.4, publierait un attribut *Fermé* et le service *Ouvrir* aurait comme précondition **require** *Fermé* ;
- Avec la sémantique du contrat concurrent, la granularité d'exécution est l'exécution d'un service (un service n'est pas interruptible). Il n'est donc pas possible d'exprimer des synchronisations ou du parallélisme au sein même de l'objet. Cette contrainte provient du fait que le contrat d'un objet est exprimé par la logique du premier ordre, qui est par essence séquentielle. Une voie de recherche possible serait d'exprimer le contrat d'un objet par des assertions exprimées en logique temporelle [Galton 87], qui souvent proposée en alternative aux RdP pour la modélisation des systèmes concurrents.

Chapitre IV. Systèmes d'objets coopératifs et organisation des classes

Le chapitre précédent était dévolu à la description des classes d'Objets Coopératifs. Le chapitre présent a pour objectif de montrer comment un ensemble de classes peut être utilisé pour la définition d'un système exécutable (IV.1), et comment cet ensemble de classes peut être structuré suivant les hiérarchies d'héritage (IV.2) et de composition (IV.3).

1. SYSTEMES D'OBJETS COOPERATIFS

Dans le formalisme des Objets Coopératifs, la seule unité de conception est la classe d'objets : le formalisme n'offre aucune construction d'un niveau supérieur, telles par exemple que les méta-classes que l'on rencontre dans la plupart des langages à objet interprétés. Toutefois, une classe est une construction statique, une description "en intention" d'entités actives ; le même ensemble de classes peut générer des ensembles différents d'instances, et il existe entre classe et instances une distinction comparable à celle qui existe entre le texte d'un programme informatique et les processus qui sont des occurrences de l'exécution de ce programme. Il faut donc donner au concepteur le moyen de définir, à partir des classes dont il dispose, un système exécutable d'Objets Coopératifs.

Un des objectifs des modèles Orientés-Objet est de favoriser la réutilisation d'éléments de conception ; il est donc primordial de maintenir une séparation totale entre la définition d'une classe et celle d'un système où cette classe est utilisée, un des critères de qualité pour la modélisation d'une classe étant son potentiel de réutilisation au sein de systèmes différents.

Un système d'Objets Coopératifs, comme tout système informatique, peut être considéré sous deux points de vue : le point de vue statique (ou structurel) décrit comment le système est construit à partir de ses constituants élémentaires, et le point de vue dynamique (ou fonctionnel) décrit les interactions que le système entretient avec son environnement. Ces deux points de vue sont détaillés dans les paragraphes suivants.

1.1. Définition d'un Système d'Objets Coopératifs.

Le mécanisme de base qui régit l'évolution d'un Système d'Objets Coopératifs (SOC) est le suivant : un objet crée un autre objet par invocation de la primitive *Create*, et obtient en résultat de cette action le nom de l'objet nouvellement créé, qu'il mémorise comme composant de la marque d'une place de son OBCS ou dans une de ses références. L'objet créateur peut ensuite demander, par l'occurrence d'une Transition d'Invocation (TI), des services à l'objet nouvellement créé, ou communiquer l'identité de ce dernier à d'autres objets du système.

Pour que ce mécanisme puisse s'amorcer, il faut qu'un (ou plusieurs) objet(s) préexiste(nt) à la mise en exécution du système. Ce sont ces **objets primitifs** qui vont, par leur activité, générer les autres objets du système. La manière dont sont créés ces objets primitifs n'est pas définie par le formalisme qui décrit le système lui-même, et le modèle ne devient une image fidèle du système réel qu'il représente qu'une fois ces objets primitifs créés et actifs.

On suppose connu un ensemble de classes d'Objets Coopératifs. Un système d'Objets Coopératifs est alors défini par un ensemble d'objets primitifs, chacun de ces objets étant lui même défini par son nom (qui l'identifie de manière unique), sa classe, et le marquage initial de son ObCS (y compris la valeur initiale de ses attributs), qui peut être différent de celui défini par l'opération *Create* de sa classe.

La plupart des langages à objet se contentent d'un seul objet primitif (souvent appelé la *racine* du système) pour définir un système, cet objet créant ensuite tous les autres ; pourquoi avoir autorisé l'existence de plusieurs objets primitifs dans un système d'Objets Coopératifs ?

Le domaine d'application du formalisme est la modélisation des systèmes concurrents. Dans une catégorie très importante de tels systèmes, les objets actifs de haut niveau sont connus en extension, et ni leur nombre ni la structure des relations qu'ils ont entre eux n'évoluent au cours de la vie du système (que l'on songe par exemple à un atelier de fabrication, ou les objets sont des postes robotisés, des tapis roulants, ... dont on connaît au départ le nombre et la répartition physique dans l'atelier). Un tel système n'utilise pas l'instanciation dynamique, au moins pour ses objets de plus haut niveau¹⁰. Il aurait paru très artificiel de définir un tel système comme un seul objet qui commence par créer tous ses composants et définir leurs inter-relations avant de les laisser à leur évolution normale. Un exemple caractéristique d'un tel système à topologie statique se trouve au chapitre VII.1.

La construction d'un système d'Objets Coopératifs se fait donc comme suit :

- 1^o étape : le concepteur désigne les objets primitifs du système, instances de classes dont il dispose. Il définit pour ces instances un marquage initial. Ce marquage peut être :
 - * Le marquage défini par l'opération *Create*. On peut alors préciser les paramètres réels à fournir à cette opération ;
 - * Un marquage initial arbitraire, différent de celui défini par l'opération *Create*, et spécifié avec la même syntaxe ;
 - * Une combinaison des deux possibilités précédentes : un marquage complémentaire est ajouté à celui défini par l'opération *Create*.
- 2^o étape : A partir de l'ensemble des objets primitifs, on peut déterminer automatiquement l'ensemble des classes des instances qui sont susceptibles d'être créées pendant l'évolution du système ; il suffit pour cela de construire la clôture transitive de la relation d'utilisation partant de l'ensemble des classes des objets primitifs. Cependant, le concepteur peut intervenir dans la manière dont cette clôture est construite : pour chaque classe examinée, il a le choix entre utiliser son implémentation ou sa spécification, ces deux descriptions étant également exécutables. Par définition, la spécification d'une classe est une feuille du graphe de la relation d'utilisation, un serveur pur : la spécification d'une classe décrit un comportement abstrait qui ne fait référence à aucune autre classe. L'implémentation d'une classe, au contraire, fera le plus souvent usage d'autres classes qui devront à leur tour être examinées dans le processus de clôture.

¹⁰La méthode HOOD est principalement destinée à la modélisation de ce genre de système, bien que cette restriction ne soit pas explicitement mentionnée dans les documents qui la décrivent.

```

System exemple
Primitive objects
  -- Trois objets primitifs, de classes différentes
  Op#1 : Classe1 is
    -- Marquage initial de l'OBCS de Op#1
    -- Défini par l'opération create de la classe
    create("Un paramètre réel");
  end -- Op#1

  Op#2 : Classe2 is
    -- Marquage initial de l'OBCS de Op#2
    -- Défini en extension : 10 jetons dans une
    -- place désignée de son OBCS
    Une_Place := 10 • <> ;
  end -- Op#2

  Op#3 : Classe3 is
    -- Marquage initial de l'OBCS de Op#3
    -- Défini en combinant les deux méthodes précédentes
    Autre_Place := 5 • <> ;
    create ;
  end -- Op#3

Classes
  -- Liste des classes qui interviennent dans la clôture de la
  -- relation d'utilisation, en précisant si on considère leur
  -- implémentation ou leur spécification.
  Classe1 : Implementation;
  Classe2 : Spécification;
  Classe3 : Spécification;

```

Figure IV.1 - Syntaxe de définition d'un système d'Objets Coopératifs

Dans l'idéal le processus de construction de la clôture des classes devrait être un processus interactif supporté par un environnement logiciel, permettant l'examen de chaque classe disponible et le choix entre son implémentation ou sa spécification. En l'absence d'un tel environnement, on décrira ici un système d'Objets Coopératifs par une notation textuelle à la syntaxe simple et auto-explicative, illustrée en Figure IV.1 : On y donne tout d'abord la description des objets primitifs, avec leur marquage initial. La clôture de la relation d'utilisation est ensuite donnée en extension, avec pour chaque classe le choix d'utiliser spécification ou implémentation.

Dans le formalisme des Objets Coopératifs, la seule unité de conception reste donc la classe, et un système est décrit avec une notation externe au formalisme. Ceci permet de bien séparer la définition d'une classe de celle du système qui l'utilise, et favorise donc la réutilisation des classes au sein de systèmes différents et l'expérimentation en cours de conception.

1.2. Le système et son environnement

Lorsqu'il entreprend la modélisation d'un système, le concepteur doit définir où se situe la frontière entre le système (qui est la partie qu'il doit décrire) et son environnement (qui échappe à sa description). Il doit également spécifier comment le système qu'il décrit interagira avec son environnement, car aucun système réel n'évolue "en vase clos".

Dans le domaine des réseaux de Petri, on rencontre fréquemment des modélisations en système clos, où l'environnement est tout simplement ignoré (un exemple classique d'une telle approche est la solution du problème des philosophes de Dijkstra, traité en §IV.3).

Une autre technique, plus réaliste, consiste à écrire un RdP comme un système ouvert, et à spécifier comment il communique avec son environnement. La communication d'un RdP avec son environnement peut être matérialisée par des **transitions sources**, c'est à dire des transitions sans arcs d'entrée [Valette...85]. Suivant la définition usuelle de la franchissabilité dans les RdP, ces transitions sont toujours franchissables. Pour modéliser l'interface avec l'environnement, on préfère définir la franchissabilité des transitions sources d'une manière "had hoc", par exemple en associant à ces transitions une fréquence de tir aléatoire, ou en les déclenchant lorsqu'une condition logique, définie à l'extérieur du RdP, est vérifiée. Cette technique est fréquemment utilisée lorsque les RdP sont utilisés à des fins de simulation, car elle permet d'étudier très facilement l'influence de l'environnement sur le comportement du système [Techlog 89].

Dans le formalisme des Objets Coopératifs, on peut utiliser une version étendue de cette technique, avec deux approches :

- La première approche consiste à considérer l'**environnement** comme un **client du système**. Certains objets primitifs peuvent être désignés pour assurer l'interface du système avec son environnement : il faut alors spécifier quels sont, parmi les services définis par leur classe, ceux qui sont susceptibles d'être invoqués par l'environnement. Ces services sont désignés sous le nom de **services d'interface**.

L'interaction entre le système et son environnement provoque alors une **action sur la structure de contrôle** du système : l'environnement peut initier des flots de contrôle, en invoquant les services d'interface.

Dans un système réel, les services d'interface serviraient à modéliser des périphériques physiques générateurs d'interruptions tels que capteurs, clavier, dispositifs de pointage, etc

Dans une optique de simulation ou de test, on peut définir avec cette approche un **scénario d'interactions**, qui spécifie quels services d'interface vont être invoqués, et quels seront les valeurs des paramètres reçus par ces services ;

- La deuxième approche consiste à considérer l'**environnement** comme un **serveur du système** ; ceci peut être fait de deux manières :
 - * L'environnement peut être défini par un ensemble de primitives, et l'interaction s'effectue dans le code des opérations internes des classes, par appel de ces primitives (Dans l'implémentation de la classe *Fichier*, décrite en §III.4.4, l'environnement est le système d'exploitation qui fournit des primitives de gestion de fichier et d'entrée/sortie) ;
 - * L'environnement peut être défini dans le formalisme Objet, par la donnée d'une **Classe Environnement** dont on ne connaît que l'interface, consistant en un ensemble d'attributs ou de services, qui sont invoqués dans les actions des transitions du système. Il existe alors un seul objet, primitif, de la classe Environnement, nommé par convention ENVIRONNEMENT.

Les techniques de temporisation décrites au §II.2.8 relèvent de ce mécanisme : l'horloge est un attribut public de l'environnement, auquel en toute rigueur on devrait accéder par la notation ENVIRONNEMENT.NOW.

La caractéristique de cette approche est que l'interaction avec l'environnement n'a pas d'action sur la structure de contrôle du système : l'environnement est complètement passif par rapport au système.

Ces deux techniques peuvent bien entendu être utilisées ensemble pour un système donné. Cette vision unificatrice de l'interaction entre système et environnement est féconde, et offre deux caractéristiques intéressantes :

- Le modélisateur peut déplacer à son gré la frontière entre le système et son environnement. Ceci est particulièrement intéressant dans une démarche d'analyse, qui commence par décrire un système dans son entier pour ensuite spécifier la composante logicielle du système ;
- Il est possible de modéliser de manière uniforme Objet, Système d'Objets et Environnement d'un Système d'Objet, ces trois entités relevant de la même description. Ceci permet, par exemple, le test unitaire d'un d'objet (considéré comme un système) en lui fournissant un environnement adéquat.

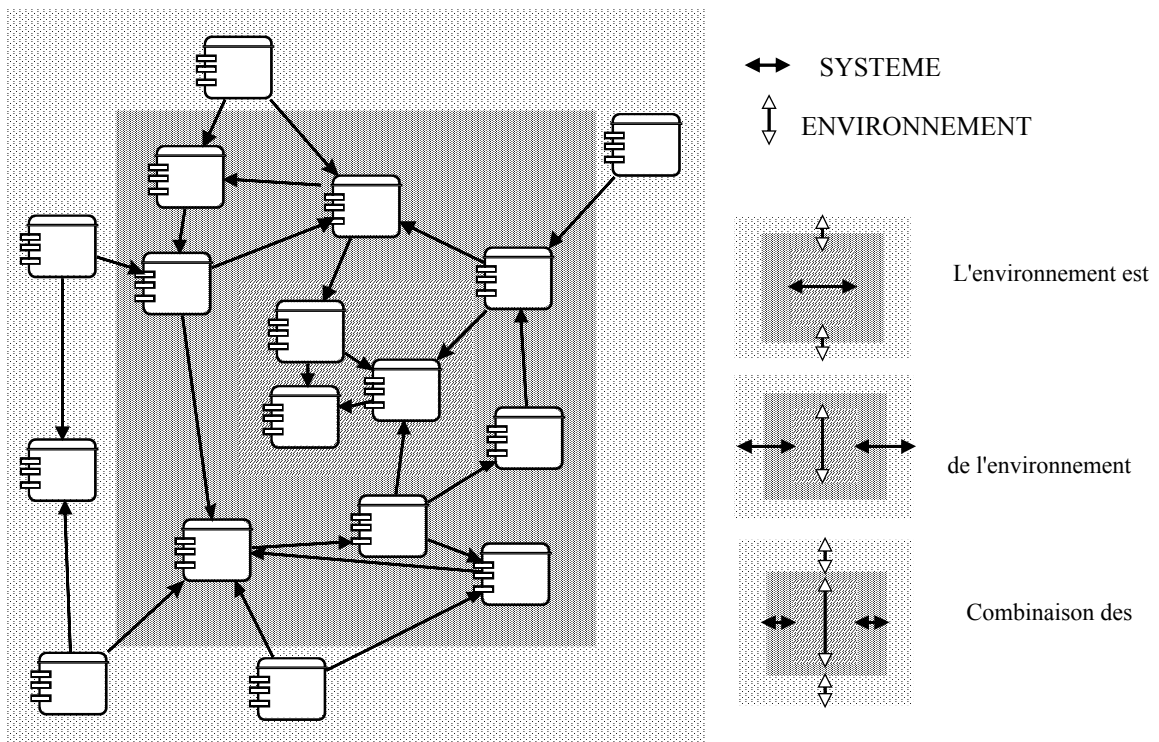


Figure IV.2 - Univers, Système d'Objets et Environnement

La Figure IV.2 illustre la vision de l'univers comme un ensemble d'objets liés par la relation d'utilisation. Suivant l'endroit où l'on place, dans cet univers, les frontières entre système et environnement on considèrera l'environnement comme un serveur, comme un client, ou comme une combinaison des deux.

Dans la troisième partie du présent mémoire, nous présentons deux études de cas utilisant notre formalisme. Une de ces études, qui décrit un atelier de fabrication robotisé, est modélisée comme un système clos. L'autre, une interface personne/logiciel, est modélisée comme un système ouvert, où l'environnement est client du système.

2. HIERARCHIE D'HERITAGE

Nous avons déjà rencontré deux types de relations qui permettent de structurer un ensemble de classes d'Objets Coopératifs :

- La première est la relation entre spécification et implémentation, qui, en donnant deux descriptions cohérentes mais distinctes d'une même classe permet de s'abstraire des détails du fonctionnement interne des instances en les considérant en temps que serveurs, ou au contraire de se concentrer sur leur activité en temps que clients ;
- La deuxième est la relation d'utilisation, qui décrit la coopération que peuvent avoir entre elles les instances de classes d'objets.

Nous allons aborder ici un autre type de relation entre classes, qui est une des caractéristiques fondamentales des systèmes à objets : la relation d'**héritage**, dont nous avons décrit les aspects fondamentaux en §I.1.4.

2.1. Héritage simple

Le formalisme des Objets Coopératifs autorise la définition de l'héritage entre classes, en lui donnant la sémantique du **sous-typage** ou de la spécialisation qui nous paraît plus appropriée dans une démarche de modélisation que la sémantique de l'**extension**, surtout nécessaire pour étendre les possibilités de factorisation et de réutilisation du code dans les langages de programmation.

La relation d'héritage sera décrite dans la spécification des classes, avec la syntaxe du langage Eiffel. Pour faire de la classe Cs une descendante immédiate de la classe Cg, il suffit d'insérer en tête de la spécification de Cs la clause :

inherit Cg ;

Une classe Cs, descendante d'une classe Cg décrit des objets **plus spécialisés** que les instances Cg. A tout instant, une instance de Cs doit pouvoir être considérée comme une instance de Cg, c'est à dire être capable de rendre les mêmes services. Un Objet Coopératif est défini par trois composantes (cf § III.3) : sa structure de données, les services qu'il offre, et son comportement. Il nous faut définir comment ces trois composantes peuvent évoluer le long de la hiérarchie d'héritage pour que cette compatibilité entre instances soit respectée.

a. Héritage et structure de données

La structure de données d'une classe est définie par un ensemble d'attributs, dont certains sont publics (déclarés dès la spécification) et d'autres privés (déclarés dans l'implémentation).

Les attributs privés d'une classe sont inaccessibles à ses clients, et donc sans intérêt pour eux. Il suffit donc pour que Cs soit compatible avec Cg en termes de structure de données que la règle suivante soit respectée :

Définition IV.1 - Héritage et structure de données

Si C_s est une classe descendante de C_g , l'ensemble A_s des attributs de C_s est inclus dans l'ensemble A_g des attributs de C_g .

Au niveau syntaxique, on s'abstiendra de répéter les attributs hérités dans l'interface d'une classe, et une classe pourra définir de nouveaux attributs publics.

Aucune hypothèse n'est faite sur les attributs privés, et deux classes reliées par la relation d'héritage peuvent avoir des ensembles d'attributs privés complètement disjoints.

b. Héritage et services

L'utilisation d'un objet se fait par l'intermédiaire de ses attributs publics, mais aussi et surtout par invocation des services publiés par sa classe.

La règle à respecter pour l'évolution des services correspond à la contrainte de sous-typage, décrite par [America 87], que nous rappelons ici :

Définition IV.2 - Héritage et services

Si C_s est une classe dérivée de C_g , alors :

pour tout service

$$s(p_1 : \text{type}_{C_{g1}}, \dots, p_i : \text{type}_{C_{gi}}, \dots, p_n : \text{type}_{C_{gn}}) : \text{type\_résultat}_{C_g}$$

exporté par C_g , alors C_s possède un service de même nom :

$$s(p_1 : \text{type}_{C_{s1}}, \dots, p_i : \text{type}_{C_{si}}, \dots, p_n : \text{type}_{C_{sn}}) : \text{type\_résultat}_{C_s}$$

tel que :

$$\forall i \in (1 \dots n) : \text{type}_{C_{gi}} \text{ est un sous-type de } \text{type}_{C_{si}},$$

et : $\text{type\_résultat}_{C_s}$ est un sous type de $\text{type\_résultat}_{C_g}$

Une classe dérivée peut donc rajouter de nouveaux services, ou accepter des paramètres plus généraux dans les services dont elle hérite. On ne donnera la signature d'un service hérité dans une classe dérivée que si cette signature évolue (cette évolution devant respecter la règle ci-dessus).

c. Héritage et comportement

Dans Eiffel, la pragmatique d'une classe est définie formellement par des pre et post-conditions portant sur les services, et par des invariants maintenus par la classe ; des règles précises définissent comment ces clauses peuvent être modifiées au sein de la hiérarchie d'héritage, afin qu'une classe dérivée puisse effectivement être considérée comme une spécialisation de ses ancêtres. Nous les avons rappelées au chapitre I, Définition I.3 (Préservation de la sémantique).

Dans le formalisme des Objets Copératifs, la pragmatique est définie par un ObCS ; il nous faut définir un critère formel de compatibilité des ObCS entre classes dérivées.

Le critère de compatibilité que nous avons retenu est identique à celui qui caractérise la compatibilité entre un ObCS de **spécification** et **d'implémentation**. Il repose sur la comparaison des langages acceptés par l'ObCS de l'ancêtre et celui du descendant, et sera détaillé au chapitre IX, avec les autres possibilités d'analyse du modèle.

Le fait que le critère de compatibilité entre ancêtre et descendant soit identique à celui entre spécification et implémentation induit une propriété importante :

la comparaison entre ancêtre et descendant peut se limiter à la spécification de classe ; si les spécifications sont compatibles et que, à chaque niveau d'héritage, les implémentations sont compatibles avec les implémentations, alors les implémentations seront également compatibles entre ancêtre et descendant ; ceci permet une bonne factorisation du processus de vérification.

2.2. Héritage multiple

On parle d'héritage multiple lorsque une classe peut avoir plusieurs ancêtres immédiats. Le formalisme des Objets Coopératifs autorise l'héritage multiple, et la relation d'héritage est donc un graphe sans cycle et non plus un arbre.

Comme dans tout formalisme autorisant l'héritage multiple, le problème du conflit de nom des caractéristiques héritées se pose (cf §I.1.4). La solution que nous avons retenue est celle du langage Eiffel, qui consiste en un renommage des caractéristiques en conflit.

Définition IV.3 - Héritage multiple et renommage des services

Soient :

- Cg1 et Cg2 deux classes qui ne sont pas en relation d'héritage,
- s un service publié à la fois par Cg1 et Cg2
- Cs une classe descendante (directement ou indirectement) de Cg1 et Cg2,

alors :

La clause d'héritage de Cs devra définir une renomination d'un des services en conflit, de la forme :

```

inherit      Cg1 rename s as <nouveau nom> ;
              Cg2

```

Une classe définie par héritage multiple doit être compatible avec chacun de ses ancêtres, en utilisant le même critère de compatibilité que pour l'héritage simple (cf. §IX.4).

3. HIERARCHIE DE COMPOSITION

Nous avons jusqu'à présent rencontré deux types de relation possibles entre classes d'objets :

- La relation d'héritage ("est-un") qui décrit le fait qu'une classe est une spécialisation d'une autre ;
- La relation d'utilisation qui décrit le fait qu'une classe a besoin des services offerts par une autre pour réaliser ses fonctions. Dans le formalisme des Objets Coopératifs, le fait qu'il existe une relation d'utilisation entre deux classes *C1* et *C2* (*C1* utilise *C2*) se traduit par le fait que *C1* possède des références vers des instances de *C2*.

Ces deux relations entre classes permettent aisément de décrire un univers d'objets atomiques, ainsi que les inter-relations entre ces objets. Toutefois, les entités du monde réel sont souvent perçues non pas comme atomiques, mais au contraire comme **composées** d'autres entités moins complexes. Un avion, par exemple, est facilement décrit comme l'assemblage d'un fuselage, de moteurs, de trains d'atterrissage,...

On voit immédiatement que la relation d'héritage ne convient pas pour décrire ce genre d'information (un avion *n'est pas* un fuselage) ; on pourrait par contre être tenté de traduire la composition par la relation d'utilisation, en considérant les objets composants comme des attributs de l'objet composé. Cette solution, qui est effectivement adoptée par de nombreux langages à objets (Eiffel, Smalltalk,...) présente plusieurs inconvénients notables [Mayer 91] :

- Elle ne permet pas de décrire des relations spécifiques qui existent entre composants et composé (par exemple des relations de type géométrique ou des contraintes topologiques) ;
- Elle encourage la confusion entre deux concepts bien distincts (utilisation et composition) qui existent dans le monde réel et qui doivent être maintenus dans le modèle que l'on en donne, surtout si l'on cherche à fournir des modèles structurés à l'image du système qu'ils décrivent.

Nous nous proposons donc d'intégrer dans notre formalisme la possibilité de décrire la relation de composition entre classes d'objets, par la notion de classe d'objets composites, ou plus simplement **classe composite** [Stefik...86]. La composition introduit la relation "est une partie de" entre classes, ou plus précisément la relation inverse "a une partie", puisque la partie n'a pas nécessairement connaissance de son tout [Blake 87].

Cette possibilité va nous permettre d'examiner un système complexe à différents niveaux d'abstraction, et non pas directement au niveau des objets atomiques qui le composent, ou de définir récursivement un objet composite en termes d'objets plus simples, jusqu'à atteindre le niveau des objets atomiques.

La notion d'objet composite est très proche de la notion d'objets parents/enfants (ou relation d'inclusion) dans la méthode HOOD ; la définition que nous allons en donner reste compatible avec

celle donnée par HOOD (cf § I.3), et un arbre de conception HOOD aura une traduction directe en termes d'Objets Coopératifs composites.

Un ensemble d'objets donne lieu à une entité composite si cet ensemble constitue un **sous-système d'objets fortement couplé** ou, selon [Stefik...86] *"un groupe d'objets interconnectés qui sont instanciés ensemble, une extension récursive de la notion d'objet. Un composite est défini par un moule (template) qui décrit ses sous-objets et leurs connexions"*. On peut caractériser un tel sous-système par les propriétés suivantes :

- Il est constitué de plusieurs instances d'objets, et le nombre de ces instances n'évolue pas au cours du temps ;
- Le graphe de la relation d'utilisation entre ces instances est constant au cours du temps.

Un tel sous-système sera modélisé comme une instance d'une classe composite.

Nous pensons fermement que la décomposition hiérarchique descendante n'est pas essentielle à la maîtrise de la complexité dans un modèle à objets. Au contraire, il semble que cette vision hiérarchique s'oppose en quelque sorte à la vision d'une organisation coopérative du logiciel, qui est à la base même des concepts mis en avant dans le formalisme des Objets Coopératifs. La décomposition hiérarchique a, en outre, souvent pour effet de réduire les possibilités de réutilisation des objets produits, car ceux-ci sont plus dépendants de leur contexte.

Aussi nous ne pensons pas que la hiérarchie de composition doive guider le processus de conception d'un système, donnant ainsi à ce processus une dynamique essentiellement descendante.

Dans un modèle d'objets coopératifs, la relation de composition sera essentiellement un **outil d'abstraction**, destiné à masquer la complexité locale d'un groupe d'objets qui coopèrent en les représentant sous la forme d'une unité fonctionnelle plus agrégée. Ainsi, la composition pourra être intégrée dans un processus de conception ascendant, permettant ainsi de fonctionner par **agrégation** plutôt que par **décomposition**. L'étude de cas du §VII.1 est un bon exemple d'une telle technique de conception.

Un objet composite est simplement constitué d'instances composantes, et n'a pas d'existence indépendamment d'elles. Les services qu'il offre à ses clients doivent donc nécessairement être des services offerts par ses instances. En général, un objet composite ne publiera qu'un sous-ensemble restreint des services offerts par ses composants, ces services représentant des opérations d'un niveau d'abstraction supérieur.

3.1. Définition d'une classe composite

Comme pour toute classe d'objets, on va fournir pour une classe composite deux descriptions : une spécification à l'usage des clients de la classe, et une implémentation décrivant la structure interne des instances.

Nous illustrerons les différents points de la définition par un exemple sans signification, uniquement destiné à mettre en lumière les conventions syntaxiques utilisées. Nous traiterons ensuite un exemple significatif, la *Table des philosophes*, où les objets composants sont les philosophes de Dijkstra.

a. spécification d'une classe composite

La spécification d'un objet composite a la même structure que celle d'un objet atomique.

Le fait d'être composite ou atomique est une caractéristique de l'**implémentation** d'une classe :

- La spécification d'une classe ne mentionne pas le fait que les instances sont atomiques ou composites. Au niveau de sa spécification, un objet composite est considéré comme un objet simple, offrant au monde extérieur un ensemble de services et publiant une description de son comportement sous la forme d'un ObCS ;
- Un objet client peut donc utiliser des objets serveurs atomiques ou composites de la même manière, en invoquant des services publiés par leur classe.

D'autre part, les trois types de relation entre classes (héritage, utilisation et composition) sont indépendants ; le mécanisme de composition que nous introduisons ici reste compatible avec les constructions que nous avons déjà présentées ; notamment :

- Un objet composite peut être créé dynamiquement de la même manière qu'un objet atomique, par appel de la primitive *Create* ;
- Un objet quelconque peut posséder des références vers un objet composite : un objet composite doit donc avoir un nom qui lui est propre, différent de celui de ses composants ;
- Une classe composite peut hériter d'une classe atomique, et réciproquement. Le fait d'utiliser la composition est donc simplement une manière particulière d'implémenter une spécification ;
- La relation de composition est récursive : un objet peut être composé d'objets eux mêmes composites.

```

Class Specification Composite_Exemple
attributes
    att1 : INTEGER;
Services
    Sv1 <P1 : Type1>;
    Sv2 <P2 : Type2> : <R : Type3>;

    create (P4 : Type4) is
        ...
    end;
obcs
    -- Description de l'ObCS de spécification
  
```

Figure IV.3 - Spécification de la classe *Composite_Exemple*

La Figure IV.3 montre la spécification de la classe que nous allons utiliser pour illustrer la syntaxe de l'implémentation des classes composites. On voit qu'aucune mention n'est faite, au niveau de la spécification, du caractère composite de l'implémentation.

b. Implémentation d'une classe composite

L'implémentation d'une classe composite doit définir comment une instance composite est constituée, et plus précisément :

- Combien d'instances composantes sont présentes dans l'instance composée, et quelles sont leurs classes ;
- Quelle est la structure de la relation d'utilisation entre instances composantes, cette structure étant déterminée par la valeur de leurs références ou par le marquage de leur ObCS ;
- Quel est l'état initial des instances composantes, c'est à dire leur état au moment où l'objet composite est créé (On veut notamment pouvoir définir pour les instances composantes un état initial différent de celui décrit par l'opération *create* de leur classe) ;
- Comment un service publié par la classe composite est traduit en une invocation d'un service publié par une des classes composantes.

Définition IV.3 - Implémentation d'une Classe d'Objets Composites

Une classe d'Objets Composites est définie par :

- l'ensemble des **objets composants**, chacun de ces objets étant décrit par sa classe et son état initial (qui comprend notamment la valeur de ses références) ;
- l'ensemble des services ou des attributs qu'elle publie ;
- Une **fonction de traduction** qui définit comment une requête sur un service ou un attribut publié par la classe composée doit être traduite en requête sur un des composants.

```

Compound Class Implementation Composite_Exemple
Attributes
    -- Les seuls attributs d'une classe composite sont des
    -- références vers les objets composants
    C1      : Classe1;
    C2, C3  : Classe2;

Translation
    -- Fonction de traduction : Renommage des attributs
    -- offerts en attributs d'instances composantes.
    att1 == C1.attC1
    -- Renommage des services
    Sv1 <P1 : Type1> == C1.SvX
    Sv2 <P2 : Type2> : <R : Type3> == C3.SvY

    -- Instanciation des composants
    Create (P4 : Type4) is
    -- 1° Création des instances composantes
    C1 := Classe1.create(P4);
    C2 := Classe2.create();
    C3 := Classe3.create();

    -- 2° Initialisation des instances composants,
    -- définit les relations d'utilisation.
    C1.SETref1(C2);
    C2.SETref2(C3);
    C3.SETref2(C2);

end;

```

Figure IV.4 - Définition textuelle de l'implémentation de la classe *Composite_Exemple*

L'implémentation d'une classe composite peut être décrite de manière textuelle ou de manière graphique. On préférera la description graphique, qui met en valeur la structure du graphe de la relation d'utilisation entre instances. Toutefois, cette notation graphique n'est qu'une facilité syntaxique, et son équivalent textuel est strictement équivalent.

La Figure IV.4 illustre la syntaxe textuelle de l'implémentation d'une classe composite, caractérisée par le mot-clé **compound** dans son en-tête :

- On définit d'abord les objets composants, qui sont des attributs-références de la classe ;
- Les services offerts sont définis dans la spécification de la classe composite, et l'implémentation donne uniquement la fonction de traduction ;
- Enfin on décrit comment l'instance composite est instanciée :
 - * On instancie tout d'abord chacun des composants ;
 - * On définit ensuite la relation d'utilisation entre ces composants ; on utilise pour cela les **services d'initialisation** qui offrent une manière standardisée de fixer la valeur initiale des attributs publics. On suppose ici que la classe Classe1 déclare un attribut public ref1 de classe Classe2, et que la classe Classe1 déclare un attribut public ref2 de classe Classe2.

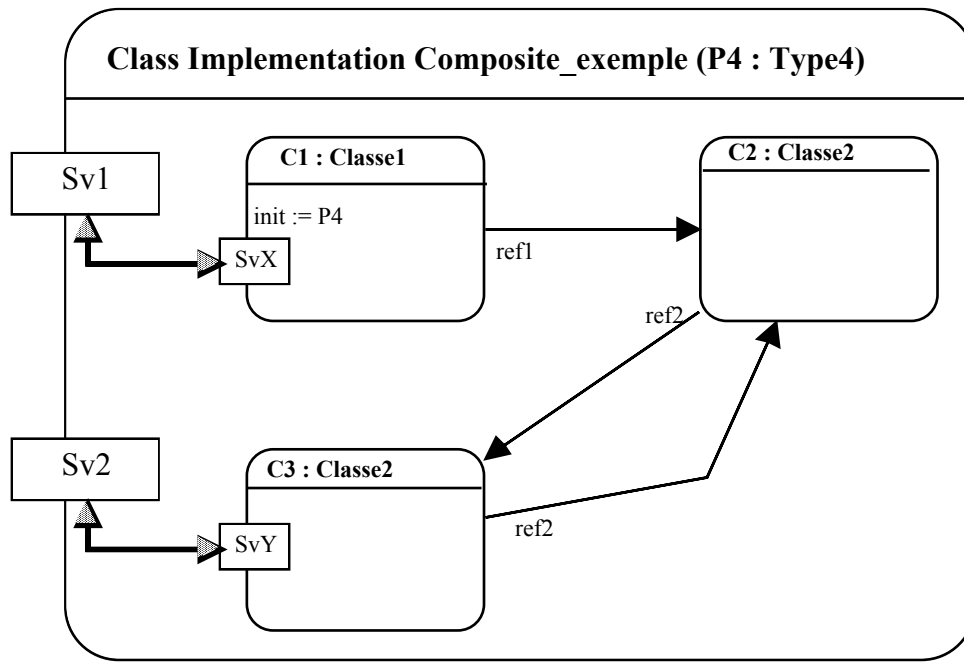


Figure IV.5 - Définition graphique de l'implémentation de la classe *Composite_Exemple*

La Figure IV.5 illustre la syntaxe graphique de l'implémentation d'une classe composite, avec les conventions suivantes, inspirées de HOOD :

- La classe composite est représentée par un rectangle englobant la définition des composants. En en-tête de ce rectangle figure le nom de la classe, et la définition des paramètres formels de l'opération *Create* de cette classe. L'interface de l'objet (les services offerts) figure dans un rectangle à la frontière du corps de l'objet ;
- Chaque instance composante est elle-même représentée par un rectangle. L'en tête de ce rectangle contient le nom de la référence correspondante dans la classe composite, et sa classe ;
- Dans le corps du rectangle figurent les attributs initiaux de l'instance composante, c'est à dire les paramètres qu'il faut passer à l'opération *Create* pour initialiser le composant. On donne le nom du paramètre formel de l'opération *Create*, et la valeur correspondante. Cette valeur peut être une constante, ou un paramètre de l'opération *Create* de la classe composite. Dans l'exemple, l'opération *Create* de la classe *Classe1* a pour signature :

```
Create(init : Type1);
```
- Les arcs qui connectent les rectangles matérialisent la relation d'utilisation entre instances : par exemple l'arc étiqueté *ref1* entre l'instance *C1* et l'instance *C2* se traduit par l'invocation du service d'initialisation *C1.SETref1<C2>*;
- La fonction de traduction est matérialisée par des arcs grisés qui connectent l'interface du composant avec les interfaces des composés.

3.2. Exemple de classe composite : la table de philosophes

Nous allons illustrer la syntaxe de définition d'une classe d'objets composites en traitant le problème des philosophes [Dijkstra 71]. Cet énoncé bien connu est un point de passage obligé pour tout modèle visant à décrire des systèmes concurrents, et l'élégance de la solution que l'on en donne permet de juger le degré d'adéquation d'un formalisme à ce genre de problème.

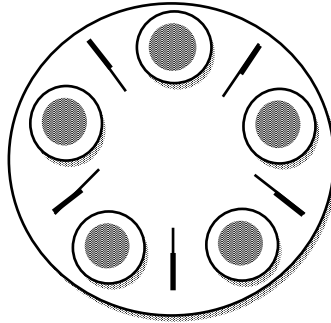


Figure IV.6 - Configuration d'une table de philosophes

Nous rappelons brièvement l'énoncé de ce problème : un certain nombre de philosophes japonais sont rassemblés, pour le repas, autour d'une table ronde. Entre chaque couple de philosophes se trouve une baguette. Pour pouvoir se mettre à manger, un philosophe doit arriver à se saisir des deux baguettes qui l'entourent (Figure IV.6). S'il y parvient, il conserve ses baguettes pendant un certain laps de temps, avant d'accepter éventuellement de les céder à ses voisins.

On voit qu'il s'agit là d'un problème de partage de ressources : les baguettes sont en nombre insuffisant pour permettre à tous les philosophes de manger en même temps. Il est notamment impossible à deux philosophes voisins de manger en même temps. La solution de ce problème doit assurer qu'une situation de blocage, où, par exemple, chaque philosophe conserverait indéfiniment une seule baguette, ne pourra pas se produire.

Nous allons produire la solution à ce problème en deux temps : tout d'abord nous décrivons la classe d'objets atomiques *Philosophe*, qui décrit le comportement de chacun des philosophes qui seront conviés au banquet. Nous décrivons ensuite la classe *Table*, qui est une classe d'objets composites décrivant l'assemblée des philosophes pour le repas.

a. La classe *Philosophe*

Un philosophe a un nom, et connaît son voisin de droite et son voisin de gauche, qui sont tous deux des philosophes. Les services que peut rendre un philosophe sont simplement de donner sa baguette gauche ou sa baguette droite, s'il les possède. *Manger* n'est pas un service que peut rendre le philosophe, mais bel et bien une opération interne. Quant à *Penser*, on a considéré ici qu'il s'agit de la tâche de fond du philosophe, et qu'il est capable de penser même en mangeant ; on n'a donc pas explicitement fait figurer cette activité dans son comportement.

La classe *Philosophe* (Figure IV.7) décrit un tel comportement, avec quelques ajouts visant à illustrer des points syntaxiques de la définition d'objets composites :

- on y a inclus un service *garnir_le_plat*, qui permet à un philosophe de réapprovisionner un plat de riz partagé par tous (et non modélisé). Nos philosophes étant droitiers, ils ne peuvent réapprovisionner le plat que s'ils ne tiennent pas de baguette dans leur main droite ;
- on y a inclus également les services d'initialisation *SETgauche* et *SETdroite*, permettant de faire connaître à un philosophe ses voisins.

On donne directement l'implémentation de la classe *Philosophe* ; sa spécification serait, en l'occurrence, identique à son implémentation, avec simplement l'omission des invocations (*droite.donne_baguette_gauche*, ...) dans l'ObCS, c'est à dire qu'elle ne décrirait pas comment un philosophe se procure les baguettes dont il a besoin.

Class Implementation *Philosophe***Attributes**

```
-- Deux références : les voisins de droite et de gauche
gauche, droite : Philosophe;
-- Une propriété : le nom propre du philosophe
nom : STRING;
```

Operations

```
-- Aucune opération interne
```

Initialization Services

```
SETgauche <pgauche : Philosophe> is
  -- Le voisin de gauche devient pgauche
  gauche := pgauche;
end; -- change_gauche

SETdroite <pdroite : Philosophe> is
  -- Le voisin de droite devient pdroite
  droite := pdroite;
end; -- change_droite
```

Services

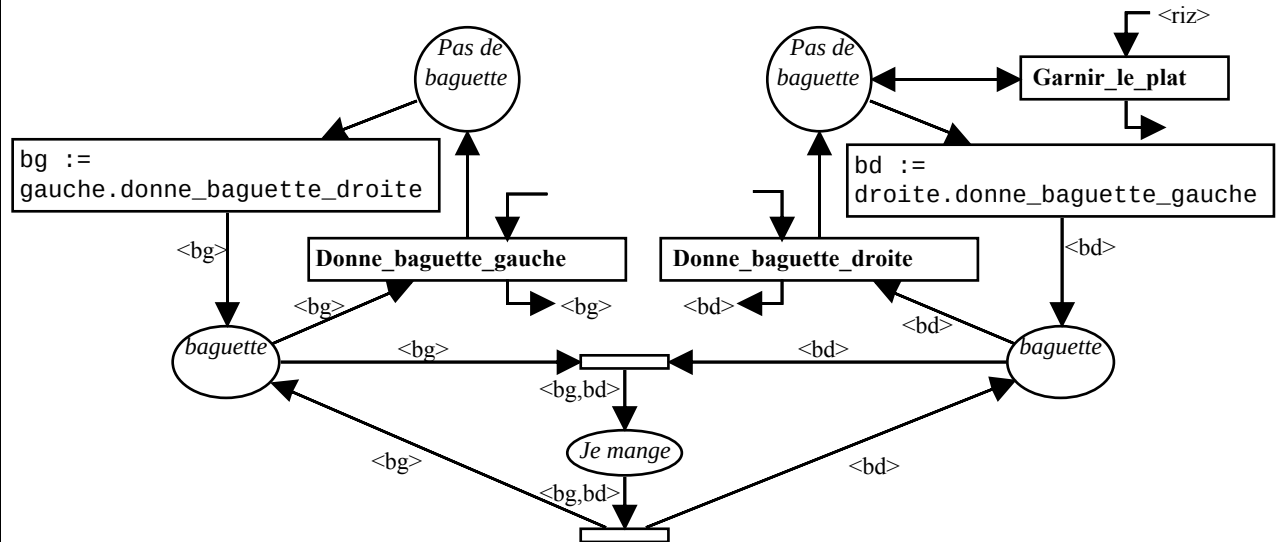
```
donne_baguette_gauche : <bg : Baguette>;
donne_baguette_droite : <bd : Baguette>;
garnit_le_plat <riz : Nourriture>;

create (pnom : STRING) is
  -- Définition des propriétés
  nom := pnom;
  -- définition du marquage initial
  pas_de_baguette_a_droite := <>;
  baguette_a_gauche := Baguette.create;

end;
```

ObCS

```
baguette_à_gauche, baguette_à_droite : <b : Baguette>;
Je_mange : <bg : Baguette, bd : Baguette>
Pas_de_baguette_à_gauche, Pas_de_baguette_à_droite : <>;
```

**Figure IV.7** - Implémentation de la classe *Philosophe*

La classe *Philosophe* ci-dessus présente un cas où le graphe d'utilisation entre classes est cyclique ; c'est même un cas d'utilisation récursive : cette classe est cliente d'elle même, puisque son ObCS contient des invocations de service à d'autres instances d'elle même (*droite.donne_baguette_gauche*, *gauche.donne_baguette_droite*).

La classe *Baguette* n'est pas détaillée : on s'intéresse ici uniquement à l'**identité** des baguettes, et les variables qui les désignent servent uniquement à spécifier le flux des baguettes entre les différents philosophes.

On trouve ici une des caractéristiques de notre approche. D'autres approches auraient modélisé la *Baguette* comme une **ressource**, en la munissant d'opérateurs tels que *Prendre* et *Reposer*. Bien que cette manière de modéliser puisse fort bien être suivie avec notre formalisme, elle se révèle ici inutile : le jeton typé, qui identifie une baguette, est suffisant pour modéliser une ressource ; l'objet dont l'ObCS contient le jeton est le possesseur exclusif de la ressource. On peut s'assurer que la ressource est toujours manipulée par **un objet et un seul** au moyen d'un critère structurel très simple sur les ObCS : il s'agit du **critère de non-duplication** [Sibertin 85] :

Définition IV.4 - Critère de non-duplication

Un RPO est dit **sans duplication** si et seulement si :

- La même variable ne figure pas sur plusieurs arcs d'entrée, ni sur plusieurs arcs de sortie d'une même transition ;
- Toutes les variables figurant sur un arc sont distinctes et ont un coefficient égal à 1.
- Un marquage est sans duplication si tout objet qui figure dans une place à une multiplicité égale à 1, et ne figure que dans cette place.

On peut montrer que si le marquage initial d'un RPO sans duplication est lui même sans duplication, alors tout marquage accessible l'est également.

Dans notre exemple, on est ainsi assuré qu'une baguette est à tout instant manipulée par un philosophe et un seul. On constate ici un double bénéfice de notre approche :

- Le modèle produit est plus simple et plus compréhensible, puisqu'il se dispense de l'introduction et de la gestion d'une classe de type "ressource exclusive" ;
- Il reste cependant complètement vérifiable, grâce au caractère formel et aux possibilités d'analyse des RPO.

Cette manière de modéliser des ressources exclusives sera reprise au §VII.1, où l'on modélise de la sorte les palettes qui circulent dans un atelier flexible de production.

b. La classe composite *Table*

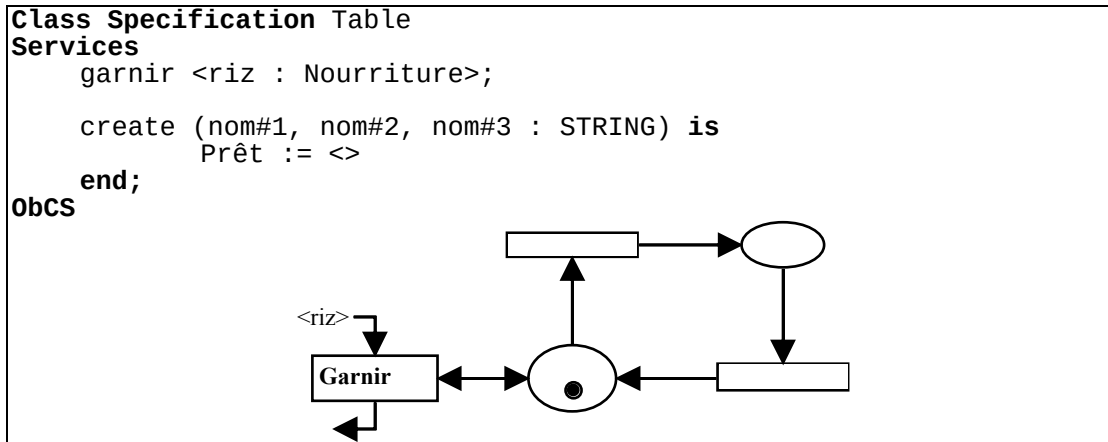


Figure IV.8 - Spécification de la classe *Table*

La classe composite *Table* modélise une table de trois philosophes. Le seul service qu'offre la table de philosophes à son environnement est la réception de riz pour le réapprovisionnement du plat ; sa spécification est décrite en Figure IV.8. Son ObCS de spécification montre seulement que le service *Garnir* n'est pas toujours disponible.

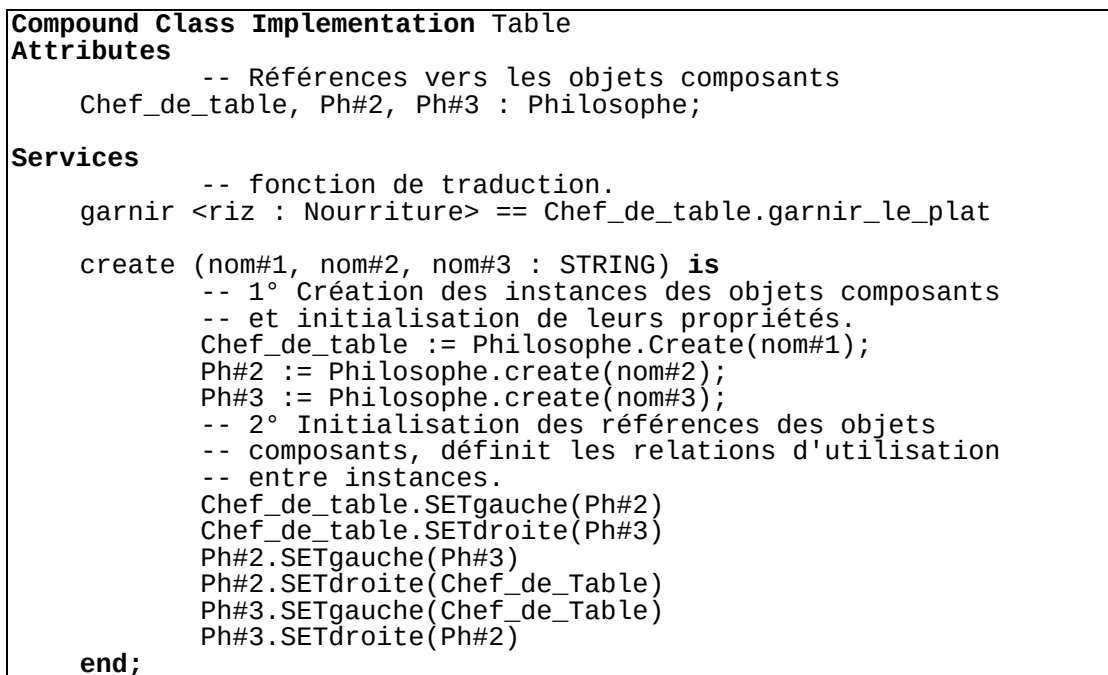


Figure IV.9 - Définition textuelle de l'implémentation de la classe *Table*

La Figure IV.9 montre la définition textuelle de l'implémentation de la classe *Table* : les seuls attributs d'un objet composite sont des références vers les objets composants ; la définition des services offerts est simplement un renommage de services offerts par des instances composantes ; la définition graphique équivalente est donnée en Figure IV.10.

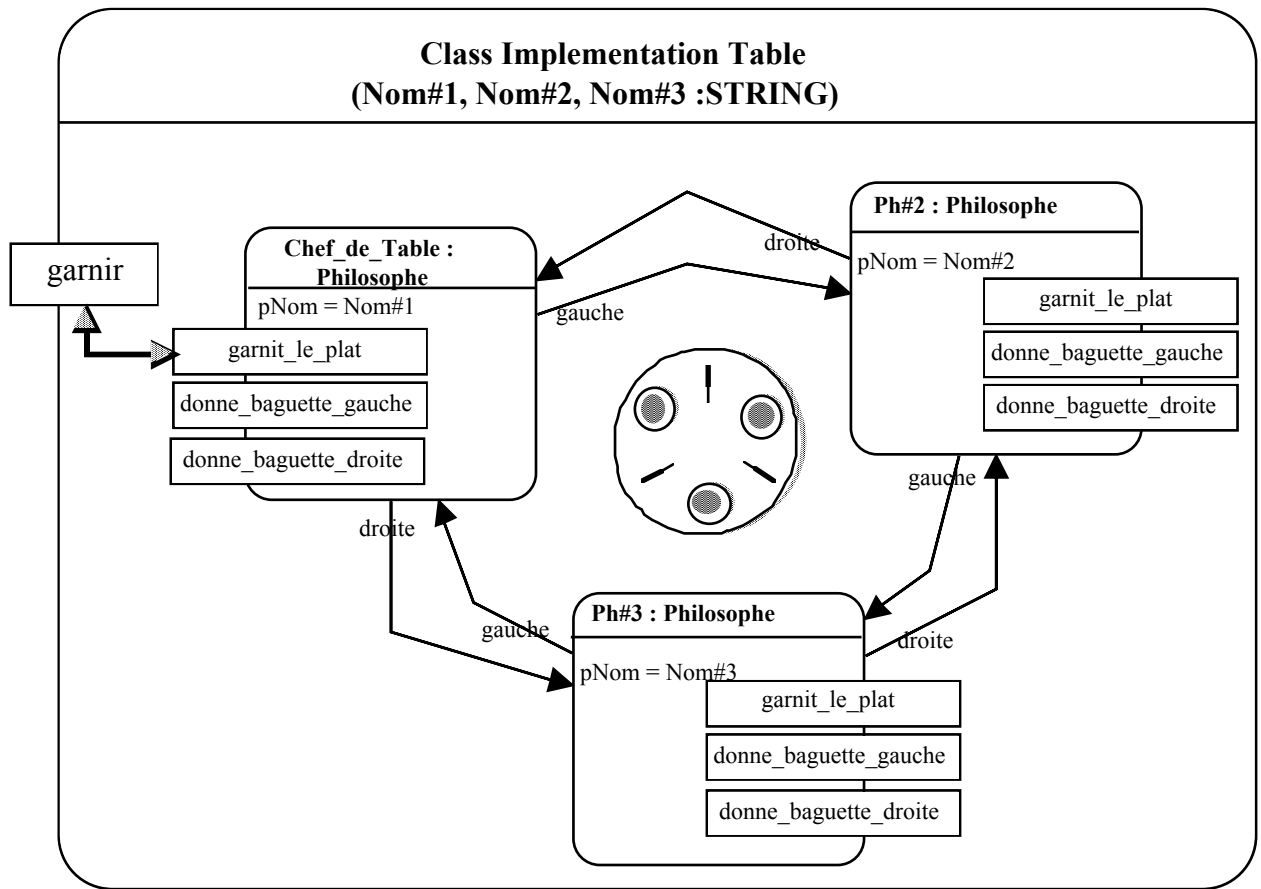


Figure IV.10 - Définition graphique de l'implémentation de la classe *Table*

Chapitre V. Sémantique formelle d'un système d'objets coopératifs

Un système d'Objets Coopératifs a un comportement non-séquentiel, et on peut y observer des flots de contrôle concurrents qui évoluent en parallèle ou se synchronisent. Intuitivement, le comportement du système doit résulter de la combinaison des comportements décrits par les ObCS des objets qui le composent, les ObCS communiquant alors par des places partagées. Le modèle d'exécution d'un système d'Objets Coopératifs est alors identique à celui d'un seul Réseau de Petri à Objets.

Cette vision intuitive se heurte à l'aspect très dynamique d'un système d'Objets Coopératifs, cette dynamicité résultant de l'instanciation dynamique et des relations entre objets :

- Instanciation dynamique : de nouveaux objets peuvent être créés, d'autres peuvent disparaître pendant l'activité du système, et l'ensemble des réseaux qui communiquent ne peut donc pas être déterminé de manière statique ;
- Relations entre objets : la relation d'utilisation entre instances ne présente pas une topologie statique, au contraire de la relation d'utilisation entre classes : les objets se connaissent par l'intermédiaire de leurs références, et ces références peuvent changer de valeur au cours du temps. Cette propriété est parfois référencée dans le domaine de la programmation concurrente sous le nom de *topologie dynamique de processus*. Cette propriété a une conséquence importante quand à l'architecture des ObCS : chaque occurrence d'une TI ne référence pas nécessairement le même objet serveur ; même la classe de l'objet serveur référencé ne peut être déterminée statiquement : elle peut varier suivant la hiérarchie d'héritage au cours des appels, en conformité avec la règle du polymorphisme.

On a défini en §III.3.5 et §III.4.5 comment un ObCS doit être interprété intuitivement par le concepteur afin que sa sémantique soit correctement comprise. Ce chapitre a pour but de décrire formellement au sein de la théorie des RPO la règle d'évolution des ObCS. Ceci est réalisé en montrant comment on peut rassembler les ObCS des classes qui composent un système en un seul réseau qui décrit la structure de contrôle du système global. Ce réseau sera appelé **WSCS**, acronyme pour l'anglais **Whole System Control Structure**. Ce mécanisme de construction décrit également comment le marquage initial du WSCS est défini à partir de l'état des objets primitifs du système.

Le WSCS que l'on génère est en fait un réseau **Prédicats / Transitions** [Genrich 87], c'est à dire que tous les jetons qui circulent sont des constantes, et non plus des références d'objet. Toute la structuration qui résulte de l'approche objet est en quelque sorte "remise à plat" par le processus de construction du WSCS.

Le WSCS définit formellement la sémantique des Objets Coopératifs dans le sens suivant :

Le comportement d'un système d'Objets Coopératifs est exactement le comportement du WSCS de ce système.

Nous aurions pu tout aussi bien définir le formalisme des Objets Coopératifs en partant de la notion de WSCS, qui n'est rien d'autre qu'un réseau Prédicat / Transitions vérifiant un certain nombre de contraintes structurelles telles qu'elle permettent de "reconnaître" les objets du système et leurs classes. Nous avons préféré à cette démarche de sémanticien une démarche orientée modélisateur.

1. GENERATION D'IDENTIFICATEURS

Afin de pouvoir gérer la dynamicité de l'évolution du système, il est nécessaire que chaque objet puisse générer des identificateurs qui soient uniques dans toute l'étendue du système. Cette possibilité est implantée par un processus récurrent : chaque objet du système a une identité unique, et peut donc générer un identificateur globalement unique en concaténant à son propre nom la valeur d'un identificateur localement unique. A cette fin, on définit un type simple appelé *Identificateur* ; chaque objet possède un attribut privé de type *INTEGER* (noté *#i*) initialisé à 0, et une opération interne appelée *gensym* en référence à son équivalent dans le langage LISP [Winston 81] dont le code est le suivant :

```
gensym : Identificateur is
  -- Génère un identificateur unique pour tout le système.
  do
    #i := #i + 1;
    -- self11 dénote l'identité de l'objet qui exécute
gensym
  result := concatener(self, #i);
end; -- gensym
```

Figure V.1 - Génération d'identificateurs

Plusieurs définitions sont possibles pour le type *Identificateur* et la fonction *Concaténer*. Nous avons choisi la solution suivante :

Type *Identificateur* : *STRING*

Concaténer : *Identificateur* x *INTEGER* → *Identificateur*

renvoie des identificateurs de la forme <nom d'instance#numéro d'ordre>

¹¹Pour dénoter l'identité de l'objet, on a préféré utiliser le mot-clé *self*, utilisé dans la majorité des langages à objets, plutôt que la désignation de la syntaxe Eiffel, qui est *Current*.

2. PRESENTATION DE L'EXEMPLE

Nous allons illustrer le procédé de construction du WSCS à partir d'un système très simple. Les classes qui composent ce système n'ont pas de fonctionnalité bien définie, mais sont construites afin de mettre en exergue tous les points qui interviennent au cours du processus de construction.

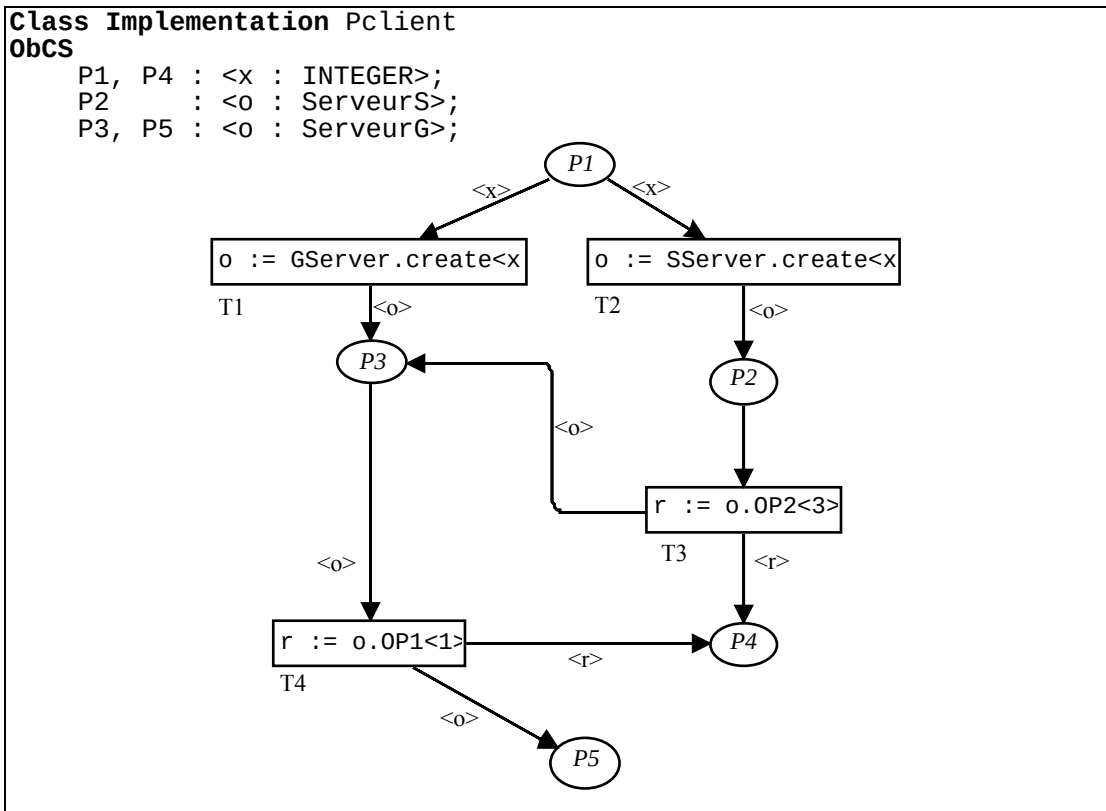
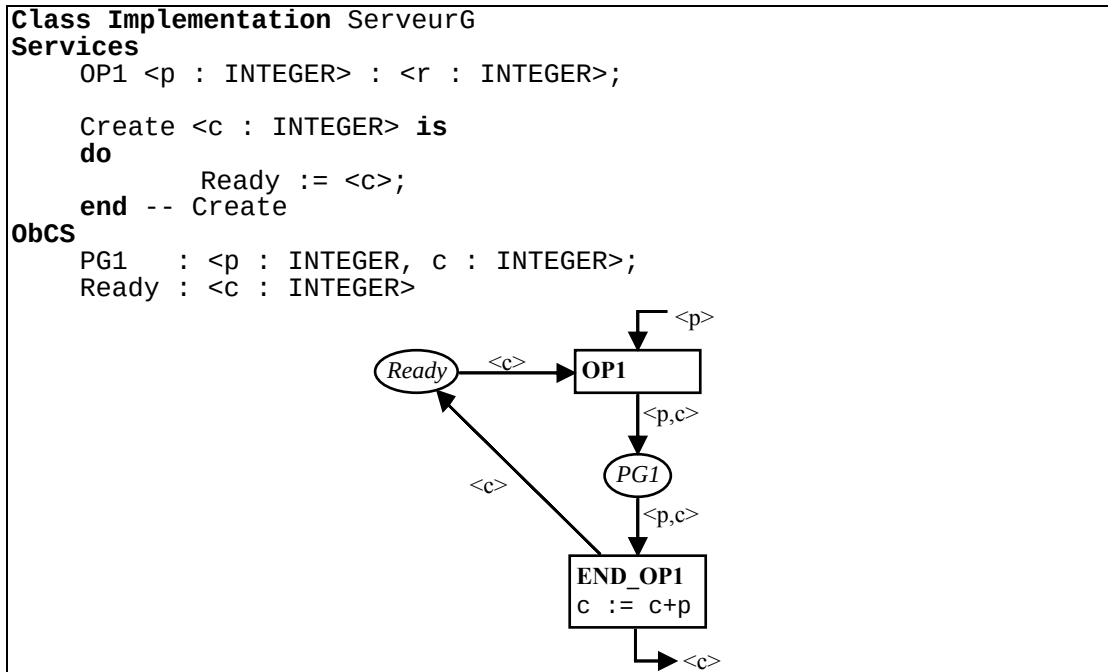
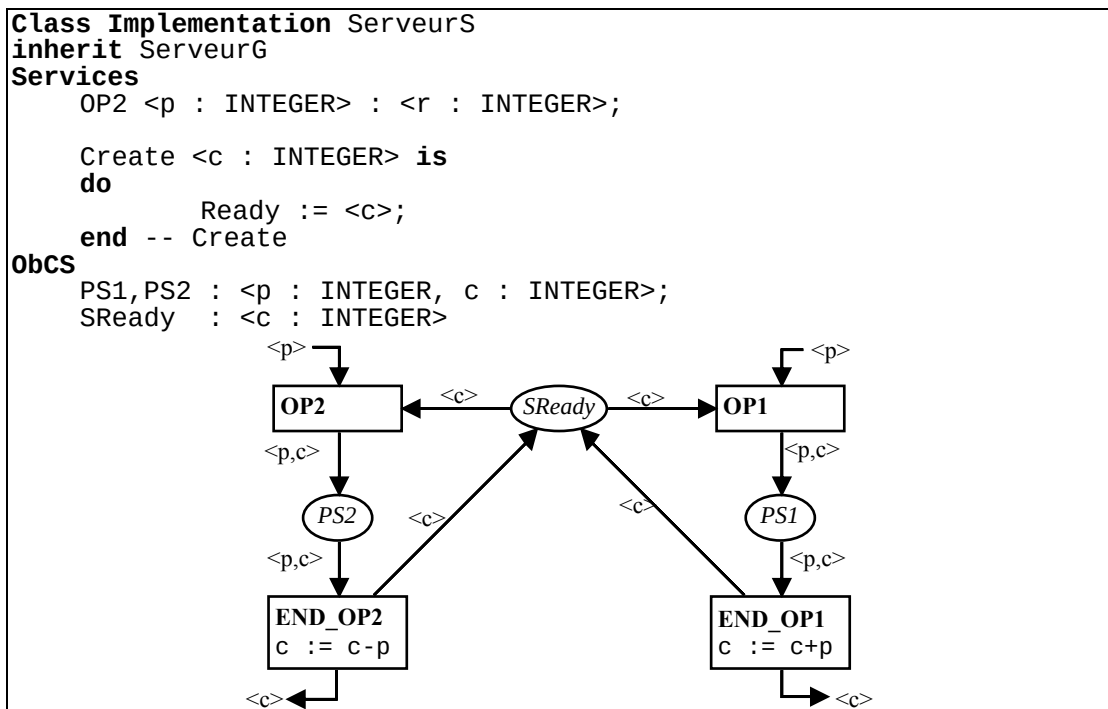


Figure V.2 - La classe *Pclient*

La classe *Pclient* (Figure V.2) décrit le comportement d'objets qui agissent uniquement en tant que clients : aucun service n'est défini pour la classe, et les instances ne peuvent donc pas être utilisées en tant que serveurs. La classe *Pclient* est cliente des classes *ServeurG* (pour serveur générique) et *ServeurS* (pour serveur spécialisé), qui hérite de *ServeurG*. Conformément à la règle d'héritage, *ServeurS* offre les mêmes services que *ServeurG* (ici uniquement *OP1*) augmentés de ses propres services (ici *OP2*).

Figure V.3 - La classe *ServeurG*Figure V.4 - La classe *ServeurS*

Une instance de *Pclient* crée de manière aléatoire des instances de *serveurG* (par occurrence de la transition *T1*) ou des instances de *ServeurS* (par occurrence de la transition *T2*). Les instances de *ServeurS* reçoivent par occurrence de la transition *T3* une requête sur leur service *OP2*, et le résultat

renvoyé est stocké dans la place *P3*. Chaque objet créé par *T1* ou *T2* reçoit une requête sur son service *OP1* par occurrence de la transition *T4*.

Les classes proposées en exemple permettent donc d'illustrer les points suivants :

- Création dynamique de nouvelles instances par occurrence de *T1* ou *T2* ;
- Relation d'utilisation dynamique entre instances : une instance de *PClient* peut utiliser plusieurs instances différentes de *ServeurG* et *ServeurS* ;
- Requête de service non polymorphique : par construction, une occurrence de la transition *T3* ne peut concerner que des instances de *ServeurS* ;
- Requête de service polymorphique : une occurrence de la transition *T4* peut concerner indifféremment une instance de *ServeurG* ou *ServeurS*.

Par contre, elles n'illustrent pas l'accès aux attributs privés dans l'ObCS d'un objet, ni l'accès à des attributs publics par des objets clients. Ces cas seront traités par de petits exemples "ad hoc" au cours de la discussion. Le traitement des classes composites, quant à lui, est exposé en §V.6

```

System exemple
Primitive objects
  -- Deux objets primitifs, Pclient#1 et Pclient#2, de classe
Pclient
  Pclient#1 : Pclient is
    -- Marquage initial de l'ObCS de Pclient#1
    -- quelques jetons dans la place P1
    P1 := <4>, <5>, <12>
  end -- Pclient#1

  Pclient#2 : Pclient is
    -- Marquage initial de l'ObCS de Pclient#2
    P1 := 2•<1>, <12>
  end -- Pclient#2

Classes
  -- Liste des classes qui interviennent dans la clôture de la
  -- relation d'utilisation, en précisant si on considère leur
  -- implémentation ou leur spécification.
Pclient : Implementation;
ServeurG : Implementation;
ServeurS : Implementation;

```

Figure V.5 - Définition du système-exemple

On s'est dispensé de définir une opération *Create* pour la classe *Pclient* : en effet l'état initial de notre système consiste simplement en deux instances de *PClient*, et aucune opération de création dynamique n'intervient sur cette classe. Dans notre exemple, les deux objets primitifs reçoivent chacun un marquage initial consistant en quelques jetons dans la place *P1*, afin que les transitions *T1* et *T2* soient initialement franchissables pour chacun de ces objets primitifs.

La définition du système correspondant à notre exemple est donnée en Figure V.5.

3. CONSTRUCTION DU RESEAU GLOBAL

La construction du WSCS est un procédé complexe, mais complètement automatisable et ne nécessite aucune intervention du concepteur, à la manière de l'édition de liens d'un programme informatique. Son principe, quant à lui, est relativement simple :

- Les RPO des ObCS des classes sont augmentés, afin de prendre en compte les divers aspects liés à la dynamique du système. Les matrices d'incidences sont modifiées, par ajout de places, d'arcs et par l'adjonction de nouvelles variables aux arcs ;
- Les ObCS ainsi augmentés sont rassemblés par fusion de places introduites par les enrichissements.

Le procédé de construction du WSCS est présenté ici en six étapes, chacune traitant un problème bien spécifique :

- L'étape 1 traite uniquement les Transitions d'Invocation (TI) ;
- L'étape 2 s'occupe de l'exécution correcte des services, côté serveur ;
- L'étape 3 traite de l'instanciation dynamique. Ces trois premières étapes procèdent à des augmentations des ObCS ;
- L'étape 4 traite de la définition du marquage initial du WSCS ;
- Les étapes 5 et 6 procèdent à des fusions de places afin d'obtenir un seul réseau global.

Etape 1 : Gestion des demandes de service.

Le but de cette étape est d'étendre les ObCS des classes clientes, afin de leur permettre d'effectuer des requêtes de service et d'obtenir les résultats de ces requêtes.

C'est dans cette étape que la possibilité d'occurrences simultanées des TI doit être prise en compte. Chaque occurrence d'une requête de service sera estampillée par le client, lui permettant ainsi de différencier les valeurs renvoyées par ses appels. On trouve ici la première utilisation du mécanisme de génération d'identifiants du §V.1.

Plus précisément, la gestion des demandes de service nécessite d'étendre les ObCS de chaque classe cliente du système de la manière suivante :

- 1° Chaque Transition d'Invocation (TI) est éclatée en une **transition d'appel**, qui a les mêmes places d'entrées que la transition originale, et une **transition de complétion**, qui a les mêmes places de sortie que l'originale ;
- 2° Pour chaque TI, une **place d'attente** est introduite afin de connecter la transition d'appel et la transition de complétion. Le type de cette place est l'agrégation des types des variables d'entrée de la TI, ou un jeton simple s'il n'y a pas de variable d'entrée ;
- 3° Une **Place paramètre** est ajoutée en sortie de chaque transition d'appel. Le type de cette place est identique à la signature d'entrée du service appelé. L'arc qui connecte la transition d'appel

avec la place paramètre est étiqueté avec les variables d'entrée de la transition qui servent de paramètres réels d'entrée du service ;

4° Une **Place résultat** est introduite en entrée de chaque transition de complétion. Cette place est destinée à recevoir les résultats fournis par l'appel du service. Cette place a pour type la signature de sortie du service. L'arc qui connecte cette place avec la transition de complétion est étiqueté avec les variables de sortie de la transition qui reçoivent les résultats du service ;

5° Un élément de type *Identificateur* est ajouté aux types des places paramètres, résultat et d'attente. Durant l'activité du système, le marquage d'une place résultat sera composé de tuples contenant les résultats obtenus par invocation du service, concaténés avec une **estampille d'appel**, c'est à dire un identifieur généré par l'occurrence de la transition d'appel. On ajoute aux étiquettes des arcs connectés à ces places la variable **id**, qui dénotera l'estampille d'appel.

Les requêtes de création d'objet (c'est à dire les transitions dont l'action est de la forme *variable := Class.create<n-uplet de paramètres>*) ne sont pas à proprement parler des TI, puisque elles ne s'adressent pas à un objet déjà créé. Elles sont toutefois étendues de la même façon.

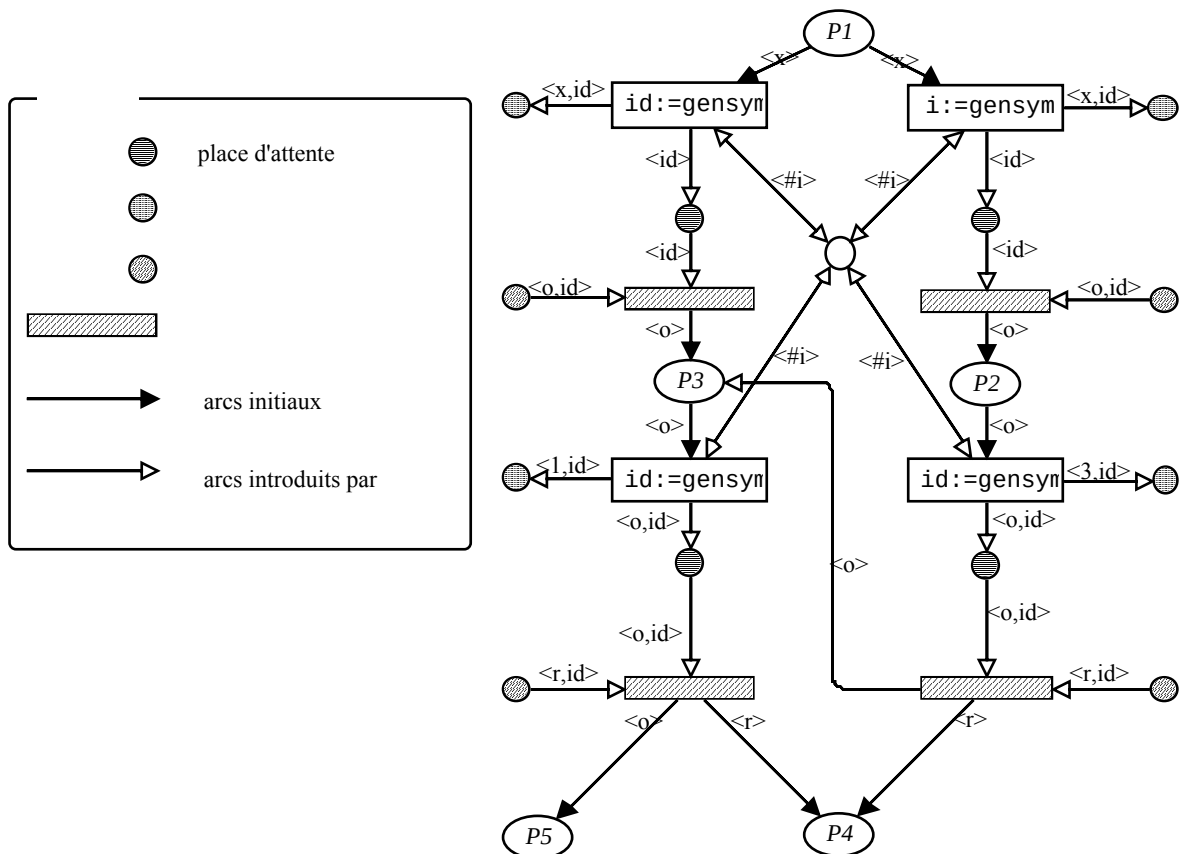


Figure V.6 - ObCS de Pclient (étape 1)

On peut voir le résultat de l'étape 1 en Figure V.6, pour l'ObCS de la classe Pclient seulement, puisque les ObCS des deux autres classes ne comportent aucune TI. Pour rendre plus explicite les transformations de structure apportées aux ObCS par le processus de construction, on a associé une représentation graphique particulière aux places et aux arcs ajoutés (explicitée en légende).

L'usage des estampilles d'appel a déjà été décrit dans [Sibertin 87] pour gérer les calculs récursifs au sein des RPO.

Etape 2 : Gestion de l'exécution des services.

Cet étape a pour but de permettre à chaque serveur de recevoir des requêtes de service avec leur paramètres, et de fournir les résultats de ces requêtes. L'ObCS d'une classe-serveur fait usage des estampilles d'appel afin de ne pas confondre les jetons qu'il manipule.

On a indiqué au § III.4.3 que l'exécution d'un service s'étend entre une transition d'accord et une transition de retour ; pour chaque classe qui joue le rôle de serveur dans le système, et pour chaque service défini dans ces classes, on doit effectuer les opérations suivantes :

- 1° Une place paramètre est ajoutée : cette place est reliée en entrée à chaque transition d'accord associée au service et les arcs qui les connectent sont étiquetés avec les paramètres formels d'entrée du service. De même une place résultat est ajoutée, reliée en sortie à chaque transition de retour associée au service et les arcs qui les connectent sont étiquetés avec les paramètres formels de sortie du service. Le type de ces places est défini de la même manière qu'à l'étape 1 ; on concatène également le type *Identifieur* aux types des places paramètre et résultat, et on ajoute conformément la variable *id* aux arcs adjacents ;
- 2° On verra au §IX que dans tout ObCS bien formé il doit exister pour chaque couple de transition accord/retour un S-invariant qui les inclut. Pour toute place contenue dans cet invariant, on concatène à son type le type *Identifieur*. On ajoute également la variable *id* à tous les arcs qui connectent les places de cet invariant.

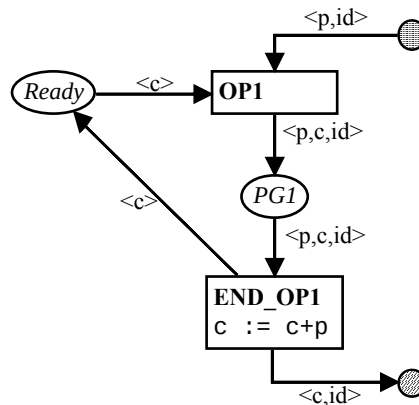


Figure V.7 - ObCS de *ServeurG* (étape 2)

Cette modification de structure produit un conflit entre transitions : au sein de l'ObCS d'une classe, toutes les transitions d'accord d'un même service sont maintenant en **conflit structurel** (cf §II.2.1.). Ceci correspond bien à la sémantique que l'on souhaite donner à l'association de plusieurs transitions au même service, à savoir montrer un indéterminisme dans le choix du traitement associé à une demande de service.

Cet indéterminisme sera d'ailleurs fréquemment levé par la structure même des ObCS (voir par exemple l'ObCS d'implémentation de la classe *Fichier_protégé*), mais le concepteur peut fort bien souhaiter conserver cet indéterminisme jusqu'à l'implémentation.

L'ObCS de la classe *ServeurG*, tel qu'il résulte de l'étape 2, est illustré en Figure V.7.

Etape 3 : Gestion de l'instanciation dynamique.

On pourrait croire, d'après le processus de construction tel qu'il a été décrit jusqu'ici, que la création dynamique de nouveaux objets dans le système va entraîner la génération de nouveaux réseaux représentant leurs ObCS, et que le système est donc modélisé par un ensemble de réseaux dont la structure évolue, d'une manière comparable à l'extension dynamique des transitions d'invocation décrite dans [Huber...89].

Nous ne voulons pas, cependant, gérer un ensemble de réseaux qui évolue dans le temps, parce que c'est renoncer à la plupart des possibilités de validation formelle liées à la théorie des RdP (toutes celles qui reposent sur l'analyse de la matrice d'incidence). Au contraire notre but est de fournir un ensemble de réseaux déterminé statiquement qui produise le même comportement résultant. L'étape 3 permet d'obtenir ce résultat en conservant un réseau pour chaque classe d'objets du système, et en rassemblant par pliage le marquage d'un nombre quelconque d'instances de cette classe.

On a indiqué au chapitre III.3 que les attributs définis pour une classe d'objet sont utilisés comme registres de son ObCS. Le chapitre II.2 a montré que les registres des RPO étaient une facilité syntaxique pour désigner des jetons présents dans des places cachées du réseau. L'ObCS de chaque classe est modifié de la manière suivante :

- 1° Introduction des places-attribut : une nouvelle place est ajoutée pour chaque attribut de la classe, et le type de cette place est le même que celui de l'attribut. Chaque place-attribut est connectée en entrée et en sortie à chaque transition où l'attribut associé est utilisé, et les arcs sont étiquetés avec le nom de l'attribut. Par utilisation d'un attribut on doit entendre :
 - Affectation d'une valeur à l'attribut au sein de l'action de la transition ;
 - Utilisation de la valeur de l'attribut au sein d'une expression ;
 - Appel d'une opération interne qui utilise dans son code l'attribut ;
 - Test de la valeur de l'attribut dans une précondition ;
 - Utilisation de l'attribut comme paramètre réel d'un appel de service.
- 2° Certaines transitions (TI ou transitions privées) peuvent n'avoir aucune place d'entrée dans l'ObCS, décrivant ainsi des services ou des actions internes qui sont toujours activables. Si le cas se présente, on ajoute à ces transitions une place en entrée et sortie, cette place possédant un marquage initial d'un jeton simple ;
- 3° Le type de la classe est ajouté à la définition du type de chaque place de l'ObCS (y compris les places qui ont été ajoutées aux étapes précédentes) à l'exception des places paramètre et résultat des TI et de places résultat des transitions de retour. On ajoute conformément la variable *self* aux arcs adjacents ;

- 4°) Le type de l'objet serveur est ajouté à la place paramètre de chaque TI. L'entité qui dénote l'objet destinataire de la requête (ce peut être une variable d'entrée de la TI ou une référence de la classe cliente) est ajoutée à l'étiquette de l'arc.

Il est important de noter que 3° et 4° donnent exactement le même type à la place paramètre en sortie d'une transition d'appel et à la place paramètre en entrée de la transition d'accord correspondante. Les places paramètres contiendront, pour chaque occurrence d'une requête de service, des jetons typés contenant les informations suivantes :

- les paramètres réels de cette invocation ;
- l'estampille d'appel générée par l'occurrence de la transition d'appel ;
- le nom de l'objet destinataire de l'appel.

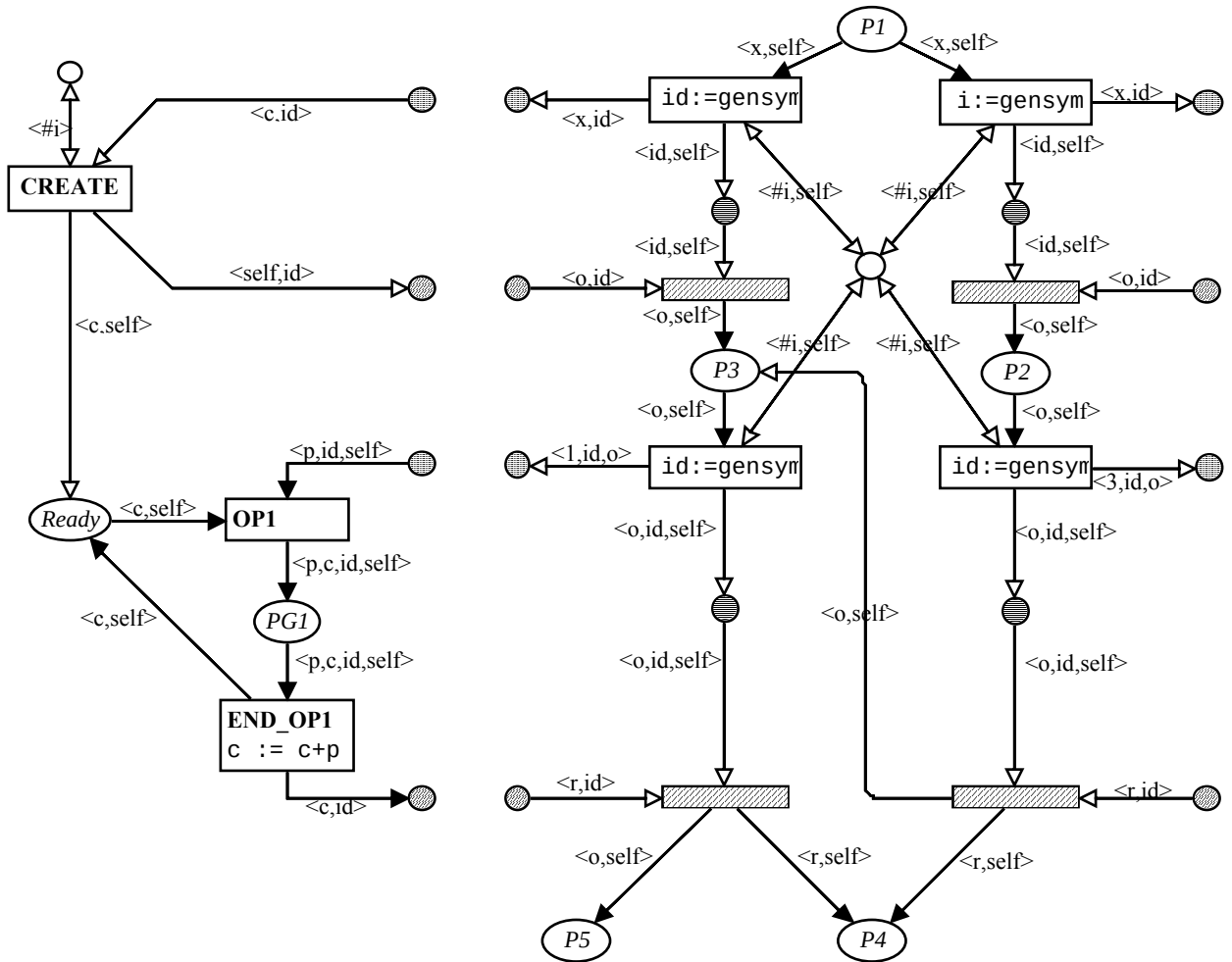
- 5° On ajoute à l'ObCS de la classe une **transition d'instanciation** avec sa place paramètre en entrée et sa place résultat en sortie. Le type de la place paramètre est la signature de l'opération *create* telle qu'elle est définie dans la classe, concaténée avec le type *Identifieur*. L'arc qui la connecte avec la transition d'instanciation est étiqueté par les paramètres formels de l'opération *create*, concaténés avec *id*. Le type de la place résultat est la classe elle-même, également concaténé avec le type *Identifieur*, et l'arc de sortie est étiqueté par *<self, id>* ;

La transition d'instanciation a un arc de sortie vers chacune des places pour lesquelles la classe définit un marquage initial non vide. Ces arcs sont étiquetés par les variables d'entrée de la transition d'instanciation ou par des constantes, de manière à ce que l'occurrence de cette transition produise le marquage initial défini dans la classe, concaténé, pour chaque n-uplet, avec la valeur de *self*.

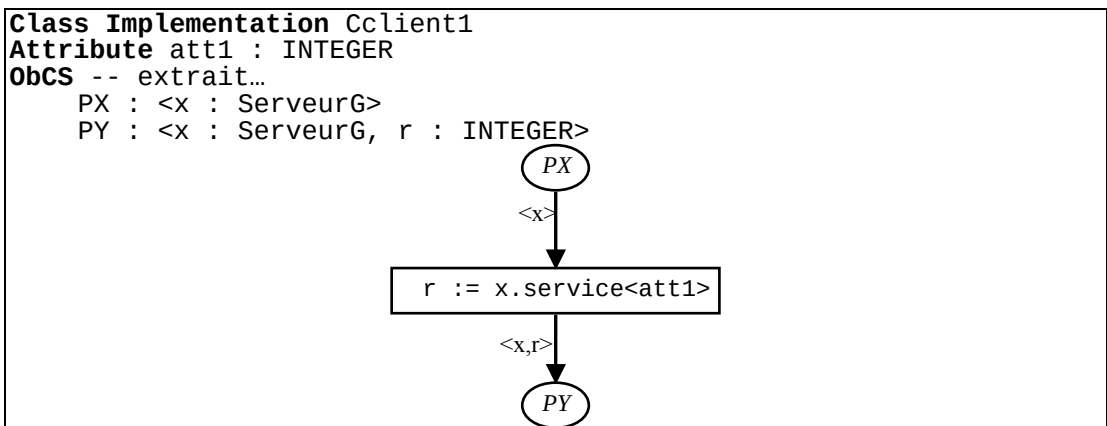
La variable *self* sera liée à un nom d'objet généré par une opération interne équivalente à *gensym*, avec pour seule différence que c'est le nom de la classe qui est concaténé avec un compteur incrémenté à chaque invocation. Ce compteur, matérialisé par un jeton stocké dans une nouvelle place connectée en entrée et sortie à la transition d'instanciation est en quelque sorte une **variable de classe**, contenant le nombre d'instances déjà générées pour cette classe. Un nom d'instance sera donc de la forme :

<Nom de classe#numéro d'instance>

Le réseau qui résulte de l'étape 3 est appelé **ObCS d'extension** de la classe, car sa structure permet de gérer l'évolution d'un nombre quelconque d'instances. On peut en voir l'exemple en Figure V.8, pour les classes *PClient* et *ServeurG*.

Figure V.8 - ObCS d'extension de *Pclient* et *ServeurG* (étape 3)

Nous illustrons par un petit exemple complémentaire le traitement des attributs, qui n'est pas couvert par l'exemple général que nous étudions.

Figure V.9 - Définition de la classe *Cclient*

Dans l'ObCS de *Cclient1*, la Transition d'Invocation *T1* invoque le service *OP1* en lui passant comme paramètre un des attributs de l'objet (*att1* de type INTEGER).

A la fin de l'étape 3, après introduction de la place-attribut *att1* et de la variable *self*, l'ObCS d'extension de *Cclient1* est celui illustré en Figure V.10.

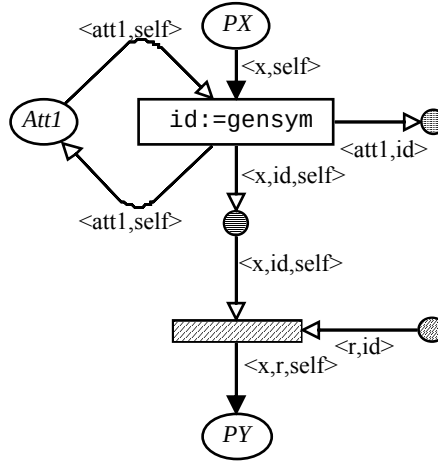


Figure V.10 - ObCS d'extension de *Cclient1*

Etape 4 : Définition des marquages initiaux.

A ce stade, nous pouvons définir pour chaque classe *C* le marquage initial *m* de son ObCS d'extension, en conformité avec l'état des objets primitifs du système.

Soit :

- *O* l'ensemble des noms des objets primitifs dont la classe est *C* ;
- *P* l'ensemble des places de l'ObCS d'extension de *C* ;
- *m_o* la distribution initiale de jetons pour *o* ∈ *O* ;
- pour tout *p* ∈ *P*, *O_p* l'ensemble des *o* ∈ *O* tels que *m_o*(*p*) ≠ 0.

Le marquage initial *m* est alors défini par :

$$\forall p \in P, m(p) = \sum_{o \in O_p} \langle m_o(p), o \rangle$$

- La place en entrée/sortie de la transition d'instanciation (utilisée pour la génération de nouveaux noms d'instance) contient initialement un jeton avec une valeur entière *<n>*, calculée pour éviter de générer des noms d'objets primitifs, et ce pour toutes les classes du système.

Le système que nous traitons consiste en deux objets primitifs, *Pclient#1* et *Pclient#2*, tous deux de la classe *Pclient*.

Ces deux objets sont munis d'un marquage initial, défini au §V.2. D'après ces marquages initiaux, le marquage de l'ObCS d'extension de Pclient est le suivant :

$P1 := \begin{matrix} <4, Pclient\#1>, <5, Pclient\#1>, <12, Pclient\#1>, \\ 2 \bullet <1, Pclient\#2>, <12, Pclient\#2> \end{matrix}$
--

Figure V.11 - Marquage initial du système-exemple

Etape 5 : Gestion du polymorphisme.

Jusqu'à ce stade, on a considéré que chaque TI concernait toujours des objets de même classe, et on n'a pas pris en compte la possibilité de polymorphisme. L'étape 5 va implémenter le polymorphisme en rassemblant en un seul réseau les ObCS d'extension des classes qui spécialisent une des classes racines de la hiérarchie d'héritage.

Ceci est réalisé en une seule opération :

- Toutes les places paramètre (resp. résultat) de chaque transition d'accord d'un service sont identifiées dans un **ensemble de fusion** (fusion set [Huber...89]) en descendant la hiérarchie d'héritage. Le type de ces places ne dépend que de la signature du service, qui est constante pour toute la hiérarchie d'héritage, et les places que l'on fusionne ont donc bien le même type ;
- Toutes les places-attribut représentant un attribut public sont elles aussi fusionnées, en descendant la hiérarchie d'héritage, créant un ensemble de fusion pour chaque attribut public ; ces nouveaux ensembles de fusion seront utilisés dans l'étape suivante, pour implémenter les TA (Transitions d'Accès à un attribut, cf § III.4.3).

Cette fusion crée un nouveau type de conflit entre transitions : ce conflit a lieu entre les ObCS des différentes classes regroupées par l'étape 5. Toutes les transitions d'accord d'un même service se trouvent en conflit le long de la hiérarchie d'héritage, puisque chacune a besoin de la marque qui contient les paramètres de l'appel, présente à chaque occurrence de l'appel dans la place paramètre résultant de l'ensemble de fusion. Ce conflit, quand à lui, ne crée aucun indéterminisme dans l'évolution du réseau : en effet, la marque paramètre de l'appel contient également le nom de l'objet destinataire, et ce nom n'est présent que dans l'ObCS d'extension d'une seule classe, celle à laquelle il appartient. Le conflit est donc résolu dynamiquement et de manière déterministe par unification sur la variable *self*. Ce mécanisme d'unification sur la variable *self* implémente la technique de **liaison tardive** (late binding) que l'on rencontre dans tous les langages à objets qui autorisent le polymorphisme.

Etape 6 : Edition de liens.

Nous sommes prêts, à ce stade, à produire un réseau unique pour tout le système, tel que chaque occurrence d'une TI dépose son paramètre dans une seule et même place, et récupère son résultat dans

une seule et même place. Le mécanisme de fusion décrit à l'étape précédente est étendu pour lier le côté client et le côté serveur de chaque requête de service, de la manière suivante :

- 1° On fusionne la place-paramètre de chaque transition d'appel avec la place-paramètre de la transition d'accord du service correspondant ; cette dernière est elle même le résultat de la fusion de l'étape précédente ;
- 2° De même on fusionne la place résultat en entrée de chaque transition de complétion avec la place résultat en sortie de la transition de retour du service correspondant.

Les étapes 1 à 3 assurent que les places que l'on fusionne à cette étape sont de même type.

Cette dernière étape de fusion crée également des conflits entre transitions : ici, ce sont les transitions de complétion du même service qui sont en conflit, toutes nécessitant une marque dans la place résultat du service. Ce conflit ne crée pas d'indéterminisme dans l'évolution du réseau, car il est lui aussi résolu dynamiquement et de manière déterministe par unification sur la variable *id*, l'estampille d'appel.

Une Transition d'Accès (TA) est, par définition, une transition qui comprend, dans sa partie action ou précondition, une expression de la forme :

`var.att1...atti...attn`, ou *var* est de la classe *Cl* et *att_i* est de la classe *Cl_i*, pour *i* : 1...n

Une telle expression est syntaxiquement correcte uniquement si :

Cl a un attribut public *att₁* de classe *Cl₁*.

Cl_i a un attribut public *att_{i+1}* de classe *Cl_{i+1}*, pour *i* : 1...n-1.

Il nous faut maintenant implanter l'accès aux attributs publics dans le WSCS.

- 3° Pour chaque TA dans les ObCS que l'on fusionne il faut :

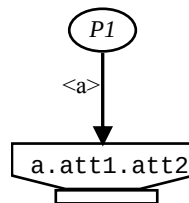
- La relier en entrée et en sortie à la place-attribut *att₁* de l'ObCS de *Cl₁*. Les arcs incidents sont étiquetés par *<att₁, var>*.
- La relier en entrée et en sortie aux places-attribut *att_{i+1}* de l'ObCS des classes *Cl_{i+1}*, pour *i* : 1...n-1. Les arcs incidents sont étiquetés par *<att_{i+1}, att_{i.}*

Comme le système-exemple que nous traitons tout au long de ce chapitre ne contient pas de TA, nous illustrons cette opération relativement complexe par un petit exemple complémentaire (Figure V.12) :

Class Implementation Cclient2

ObCS -- extrait...

P1 : <var : Cserveur1>



Class Specification Cserveur1

att1 : Cserveur2 -- Un attribut public de classe Cserveur2.

Class Specification Cserveur2 att2 : BOOLEAN -- Un attribut public Booléen.

Figure V.12 - accès chaîné aux attributs publics

L'expression `var.att1.att2`, présente dans la précondition de l'ObCS de CClient2, s'évalue bien comme un Booléen. La transition T1 est donc une TA, avec $n = 2$.

Le WSCS généré par ce fragment d'ObCS est illustré en Figure V.13.

On voit que l'accès aux attributs publics est implanté d'une manière très différente que l'invocation : on accède directement au marquage d'une place d'un autre ObCS, sans passer par l'intermédiaire du mécanisme de synchronisation réalisé par les places paramètre et résultat, et le couplage entre l'objet client et l'objet serveur en est d'autant plus fort.

Ceci motive en grande partie les réserves méthodologiques que nous avons émises en §III.3 quant à l'utilisation des attributs publics. Nous verrons en §VIII que cet accès pose également problème lorsque on veut réaliser une implantation répartie du modèle.

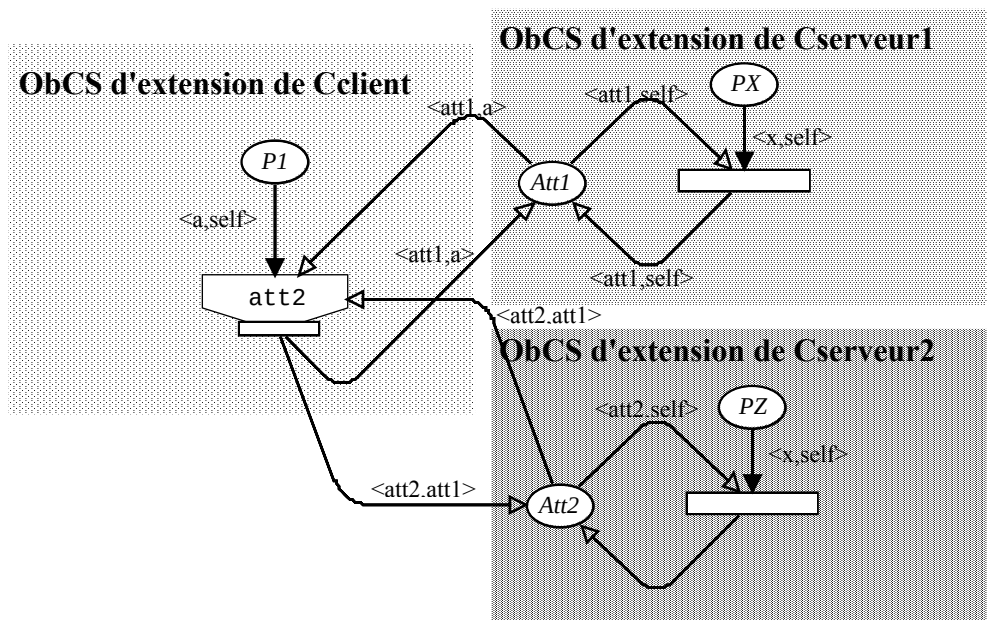


Figure V.13 - Implantation des accès aux attributs dans le WSCS

Il n'est toutefois pas douteux que l'accès à des attributs d'autres objets soit utile dans de nombreux cas de figure. Le cas traité ci-dessus en est un exemple typique : on peut évaluer un attribut dans une précondition en préservant l'atomicité de celle-ci, ce qui serait impossible si on devait passer par une invocation pour accéder à la valeur de l'attribut.

Dans le chapitre VI nous introduisons les notions de **services de consultation**, qui permettent d'étendre les possibilités d'accès aux attributs..

L'étape 6, que nous venons de décrire, termine le processus de construction, et le WSCS de notre système-exemple est illustré en Figure V.14.

Figure V.14 - WSCS du système-exemple

4. REGLE DE TIR POUR LE RESEAU GLOBAL

La règle de tir qui doit être appliquée au WSCS est la suivante : un nombre quelconque de transition peuvent être tirées au même instant, pourvu que la variable *self* reste liée à des noms d'objets différents. Cette règle de tir est conforme à la sémantique naïve des ObCS (chapitre III) qui est que plusieurs objets sont actifs au même instant, bien que chaque objet ne puisse effectuer qu'une seule opération à la fois.

5. INTERPRETATION INTUITIVE ET FORMELLE DES OBSCS

En s'appuyant sur l'exemple précédent, nous allons donner ici une justification de la correspondance entre la sémantique intuitive d'un ObCS (décrite en §III.4.5) et sa sémantique formelle. Cette justification se fonde sur un invariant du WSCS que nous venons de construire.

Pour $i \in \mathbb{N}^*$, on notera pr_i la projection d'un n-uplet sur son i° composant. On définit alors :

- $F_{P1} = pr_1$,
- $F_{Ready} = F_{SReady} = F_{P4} = \emptyset$ (fonction identiquement nulle),
- $F_{P2} = F_{P3} = F_{P5} = pr_2$,
- $F_{pi} = pr_2 - pr_1$ pour les places d'attente sur les requêtes de création d'objet,
- $F_{pi} = pr_3 - pr_2$ pour les places d'attente sur les requêtes de OP1 et OP2,
- $F_{pi} = pr_2$ pour les places paramètres et résultats places,
- $F_{PG1} = F_{PS1} = F_{PS2} = pr_3$,
- $W = (F_p)_{p \in P}$.

Alors $W.C = 0$, et W est un invariant du WSCS .

En fait, l'identificateur d'appel présent dans la place d'attente d'une Transition d'Invocation est le même que celui qui circule dans le chemin de places entre la transition d'accord et la transition de retour, et les places d'attente sont implicites [Berthelot 86] par rapport à leur composant de type Identificateur.

Lorsque une séquence de transitions peut être franchie dans un OpCS (i.e. le serveur est en mesure d'accomplir le service), les transitions de requête et de complétion de la TI sont franchies dans le WSCS de la même manière que la TI est franchie dans l'ObCS du client (avant toute expansion).

Il faut noter que le WSCS du système-exemple est **borné**. Il en est toujours ainsi si les ObCS des classes sont bornés, et si les requêtes de création d'objet se produisent en nombre fini [Sibertin 91b].

6. PRISE EN COMPTE DES CLASSES COMPOSITES

Nous n'avons pas intégré le traitement des classes composites dans le procédé de construction du WSCS décrit en §V.3, pour éviter des digressions qui auraient compromis la lisibilité de la procédure de construction. Les classes composites ne sont pas, d'ailleurs, un élément fondamental du modèle, mais essentiellement un outil de structuration et d'abstraction.

Dans ce chapitre nous allons expliciter la manière d'intégrer les classes composites dans la construction du réseau global. Nous illustrerons cette intégration en complétant l'exemple des philosophes, décrit au §IV.3.2 ; Ceci nous permettra également de démontrer la construction du WSCS sur un exemple significatif, et de voir les enseignements que l'on peut tirer de l'analyse du WSCS.

L'utilisation de classes composites introduit deux problèmes à traiter dans la construction du WSCS :

- L'instanciation d'un objet composite, plus complexe que celle d'un objet atomique;
- La traduction de requêtes vers une instance composite en requêtes vers les composants.

Ces deux problèmes sont résolus par l'introduction d'un petit réseau intermédiaire, qui se charge de l'instanciation des composants et de l'indirection des requêtes à l'instance composée vers les instances composantes ; ce RPO est appelé le **réseau de traduction** de la classe composite. Ce réseau définit notamment quelle est l'instance composante qui reçoit l'appel, et quel est le service correspondant à appeler.

Le réseau de traduction est la seule composante qui soit propre à l'objet composite : en dehors de ce réseau, un objet composite n'est qu'une enveloppe qui associe des objets composants, et n'a pas d'existence en dehors de ces composants. Le réseau de traduction est généré automatiquement, à partir de la définition de la classe composite.

Le réseau de traduction est en quelque sorte l'implémentation au moyen des RPO de la technique de **transmission de message** (message forwarding [Blake 87]), utilisée pour implanter la hiérarchie de composition dans le langage Smalltalk. Cette technique est également proche de la **délégation** des langages d'acteurs [Agha 86b] : un objet composite délègue l'exécution de ses services à ses objets composants.

Les deux sections suivantes montrent comment le réseau de traduction traite le problème de l'instanciation d'un objet composite et celui de son invocation.

6.1. Instanciation d'un objet composite

La création d'une instance composite est plus complexe que celle d'une instance atomique : elle nécessite la création des instances composantes, et l'invocation sur ces instances de **services d'initialisation** (cf §III.3.5). Cette création, qui est une séquence d'actions simples, doit cependant être perçue comme une action unique par l'objet qui l'effectue.

Une des caractéristiques d'un objet composite est que les instances qui le composent ont entre elles une relation d'utilisation statique ; or cette relation d'utilisation est définie, au niveau d'une instance, par des références qui sont contenues dans le marquage de son ObCS. Il est donc nécessaire d'avoir un moyen d'initialiser ces références, en s'assurant qu'elles ne pourront plus changer de valeur au cours de la vie de l'objet.

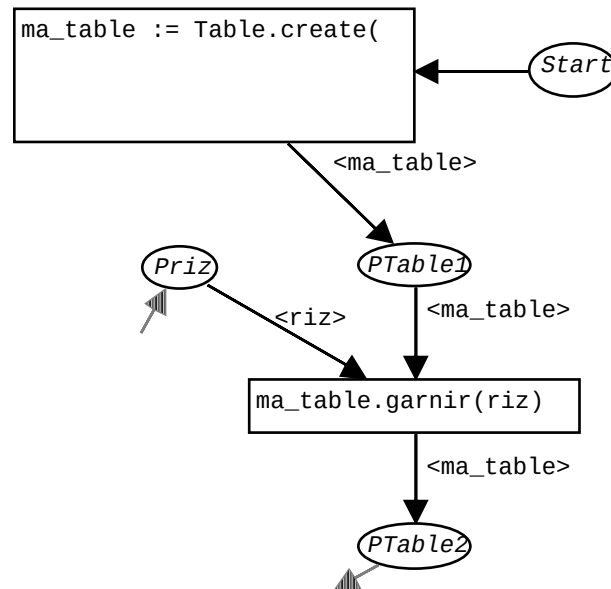


Figure V.15 - Instanciation et invocation d'une classe composite

On pourrait envisager de fournir toutes les références nécessaires en tant que paramètres de l'opération *create*, mais cette solution n'est pas satisfaisante car elle interdit les relations d'utilisation cycliques entre instances. Pour remédier à ce problème, on utilise les services d'initialisation, définis au paragraphe III.3.5, et qui offrent une manière standardisée de fixer la valeur initiale d'un attribut.

La Figure V.15 montre un client qui instancie la classe composite *Table* (§IV.3.2), et invoque ensuite l'instance composite créée.

La Figure V.16 montre la structure du WSCS qui sera généré par l'instanciation de la classe composite, en se limitant à l'ObCS du client et au réseau de traduction de la classe composite.

L'ObCS d'extension de la classe *Philosophe* sera construit et détaillé ci-après.

On voit que la classe composite mémorise le nom de ses instances composantes dans des places-attributs. On utilisera ensuite le nom de ces composantes pour rediriger les invocations reçues par l'instance composante.

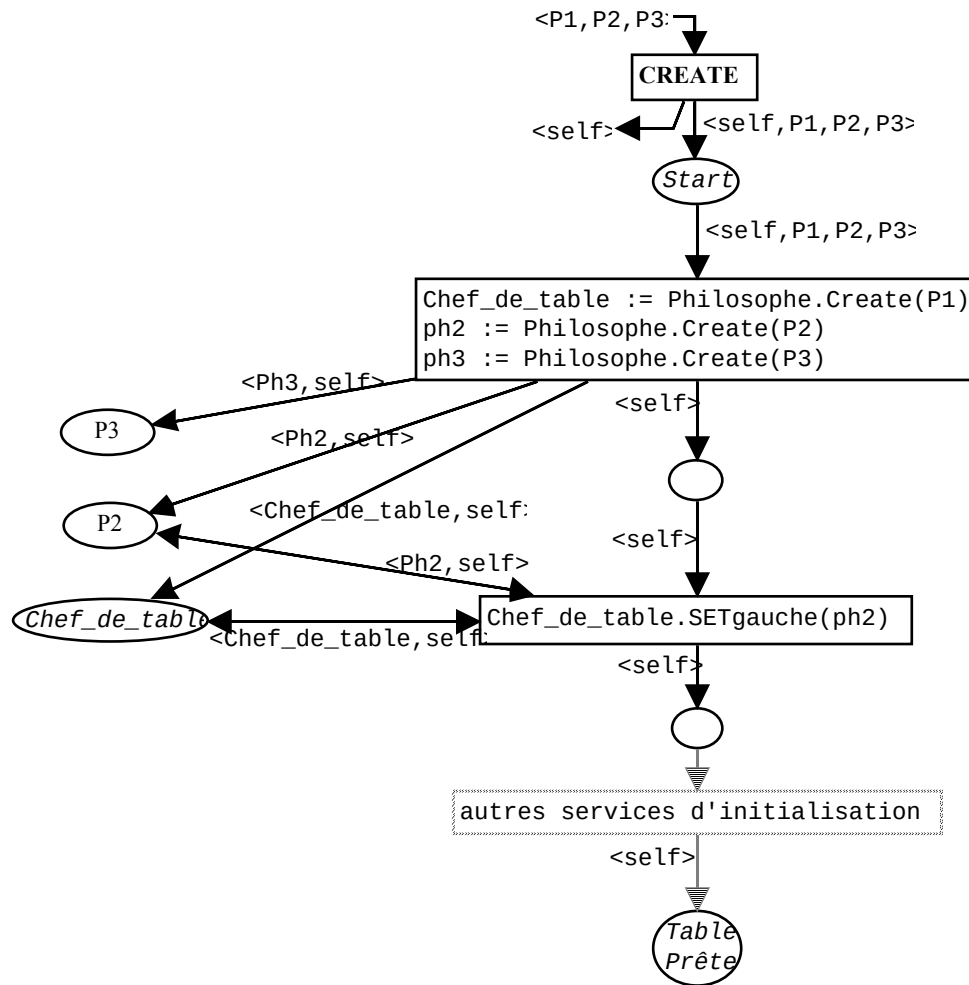


Figure V.16 - Réseau de traduction pour l'instanciation d'une classe composite

6.2. Invocation d'une instance composite

Le **nom d'un objet composite**, renvoyé par l'opération *Create*, est différent du nom de ses objets composants. L'invocation d'un service à une instance composite nécessite donc un mécanisme de décodage simple, décrit par le réseau de traduction de la classe composite, et qui permet à l'instance composite de déléguer les invocations qu'elle reçoit vers ses composants.

Ce réseau définit quelle est l'instance composante qui reçoit l'appel, et quel est le service correspondant à appeler.

Dans l'exemple de la table de philosophes, le seul service exporté par la classe composite est *Garnir*, qui se traduit en une invocation du service *Garnir_le_plat* sur l'instance composante *Chef_de_table*.

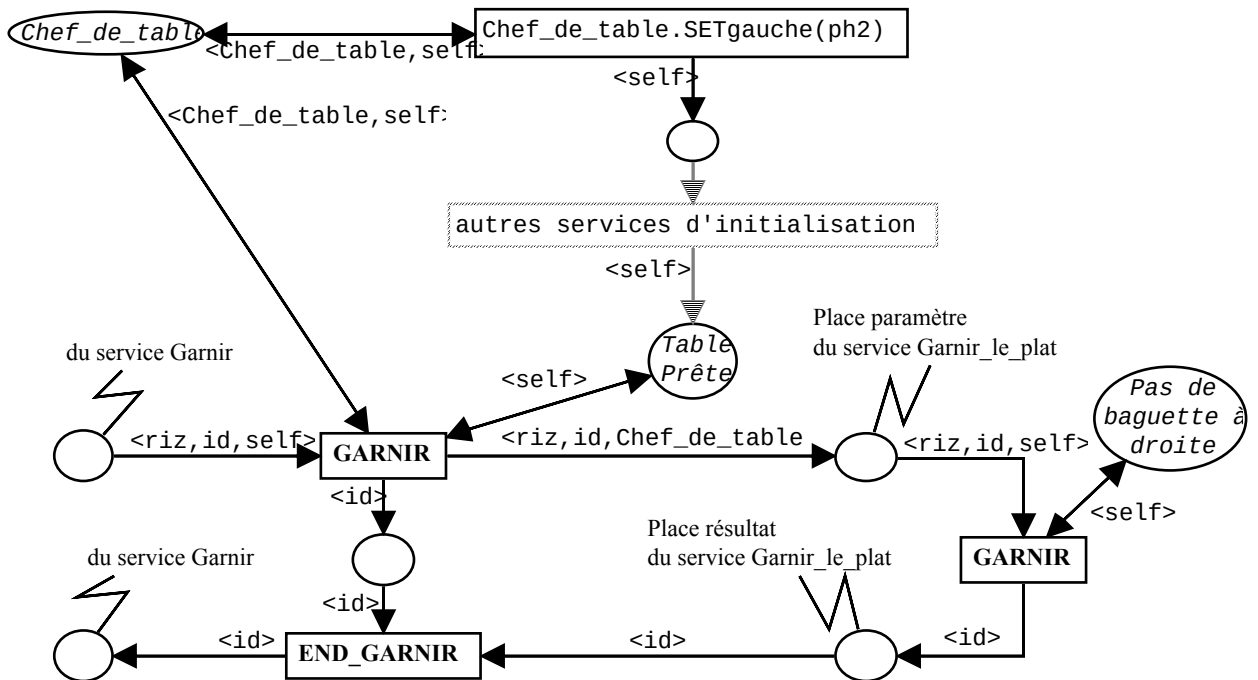


Figure V.17 - Réseau de traduction pour l'invocation d'une instance composite

La Figure V.17 montre le mécanisme qui permet la redirection de l'appel vers l'instance composante, en complétant la structure du réseau de traduction entamée en Figure V.16.

6.3. Construction d'un système à partir d'une classe composite

L'exemple que nous avons traité tout au long du §V.3 pour illustrer le procédé de construction du WSCS était sans signification, uniquement destiné à montrer l'effet des différentes étapes du procédé sur la structure des ObCS.

Nous nous proposons d'effectuer la construction du WSCS sur un système qui fasse sens, afin de démontrer les enseignements que l'on peut tirer de cette construction et nous allons traiter l'exemple de la table de philosophes, décrit au §IV.3.2.

Le système que nous allons traiter est constitué d'une seule instance de la classe *Table*. La définition du système est illustrée en Figure V.18. Une définition équivalente, qui ne fait pas appel à la classe composite mais accède directement aux classes composantes est donnée en Figure V.19. On voit que cette définition, uniquement textuelle, met beaucoup moins bien en valeur la structure des relations entre instances, et produit donc un modèle plus difficile à lire et à comprendre.

```

System Table_de_philosophes
Primitive objects
  -- Un seul objet primitif, dont le nom est Table#1
  Table#1 : Table is
    create ("Yösaï", "Myözen", "Dögen")
  end -- Table#1

Classes
  Table      : Implementation;
  Philosophe : Implementation;

```

Figure V.18 - Définition du système (version 1)

```

System Table_de_philosophes2
  -- Définition sans objets composites
Primitive objects
  -- Trois objets primitifs de classe Philosophe
  Ph#1, Ph#2, Ph#3 : Philosophe;

  Ph#1 is
    create("Yösaï");
    gauche := Ph#2;
    droite := Ph#3;
  end -- Ph#1

  Ph#2 is
    create("Myözen");
    gauche := Ph#3;
    droite := Ph#1;
  end -- Ph#2

  Ph#3 is
    create("Dögen");
    gauche := Ph#1;
    droite := Ph#2;
  end -- Ph#3

Classes
  Table      : Implementation;
  Philosophe : Implementation;

```

Figure V.19 - Définition du système (version 2)

En suivant le procédé décrit au §V.3 on peut construire le WSCS de ce système, représenté en Figure V.20, où l'on s'est simplement dispensé de faire figurer les transitions représentant les services *SETgauche*, *SETdroite*, et *garnir*. On remarque notamment les places *voisin de droite* et *voisin de gauche*, qui sont la traduction des attributs *gauche* et *droite* de la classe.

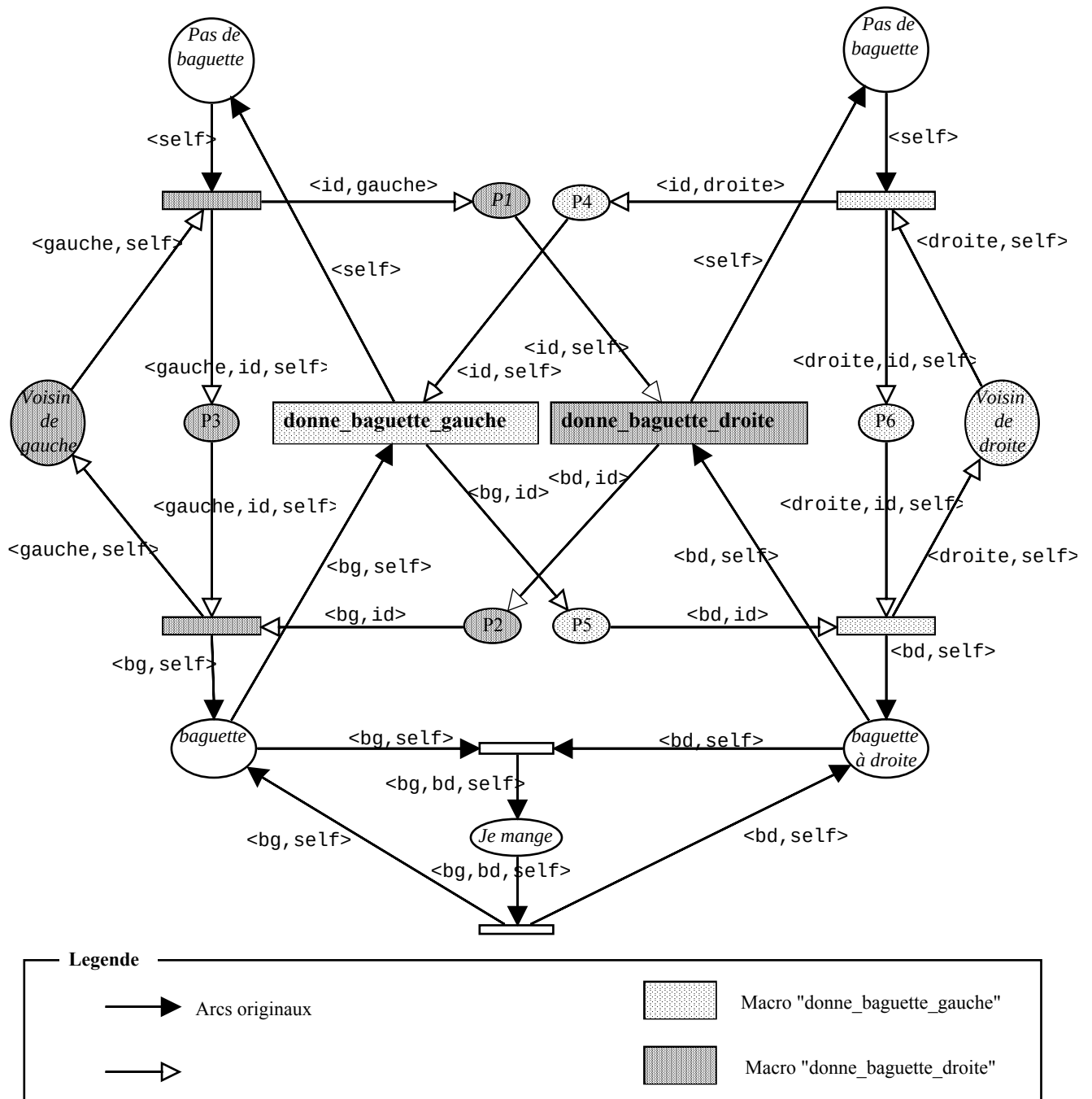


Figure V.20 - WSCS de la table de philosophes

La structure du WSCS peut paraître relativement complexe par rapport aux réseaux qui sont produits pour le même problème par une approche classique fondée sur les réseaux colorés [Jensen 81]. Il faut toutefois remarquer les avantages suivants de notre approche :

- Le travail du concepteur consiste uniquement à définir la classe *Philosophe*, qui décrit le comportement d'une seule instance, et reste donc très simple. Le travail de "repliage", qui est du ressort du concepteur dans une approche par réseaux colorés, est ici complètement automatique ;

- Le WSCS décrit le fonctionnement du système avec un grand luxe de détails ; notamment, le transfert d'une baguette entre un philosophe client et un philosophe serveur est détaillé en :
 - L'envoi d'une requête du client au serveur,
 - La prise en compte de la requête par le serveur,
 - Le retour du résultat de la requête (ici un simple jeton) au client.

On peut analyser le WSCS en calculant les invariants, qui en éclairent le fonctionnement.

- Un invariant décrit le flux des baguettes entre les philosophes. Cet invariant (Baguette_à_gauche, P5, Baguette_à_droite, P2, Baguette_à_gauche) montre qu'une baguette passe continuellement de la main gauche d'un philosophe à la main droite de son voisin ;
- L'invariant complémentaire (Pas_de_baguette_à_gauche, P1, Pas_de_baguette_à_droite, P4, Pas_de_baguette_à_gauche) décrit le flux de "l'absence de baguette".

Le WSCS peut en outre être simplifié, par simple regroupement des deux macro-transitions, soulignées en Figure V.20 par des tons de gris différents, et en identifiant les deux places *voisin de droite et voisin de gauche*, devenues redondantes, en une seule.

Ce faisant, on ne détaille plus la séquence d'opérations *demandeur une baguette, attendre la baguette et récupérer la baguette obtenue*. Il faut remarquer qu'un tel regroupement n'est possible que si un service est invoqué par une seule TI du système, et que le système est sans héritage.

Au prix d'une simple renomination des variables, on obtient alors le réseau de la Figure V.21, très proche d'une modélisation par réseaux colorés. La place *à gauche* représente alors un prédicat tel que :

$\text{à\_gauche}(X,Y)$ est vrai si X est à gauche de Y .

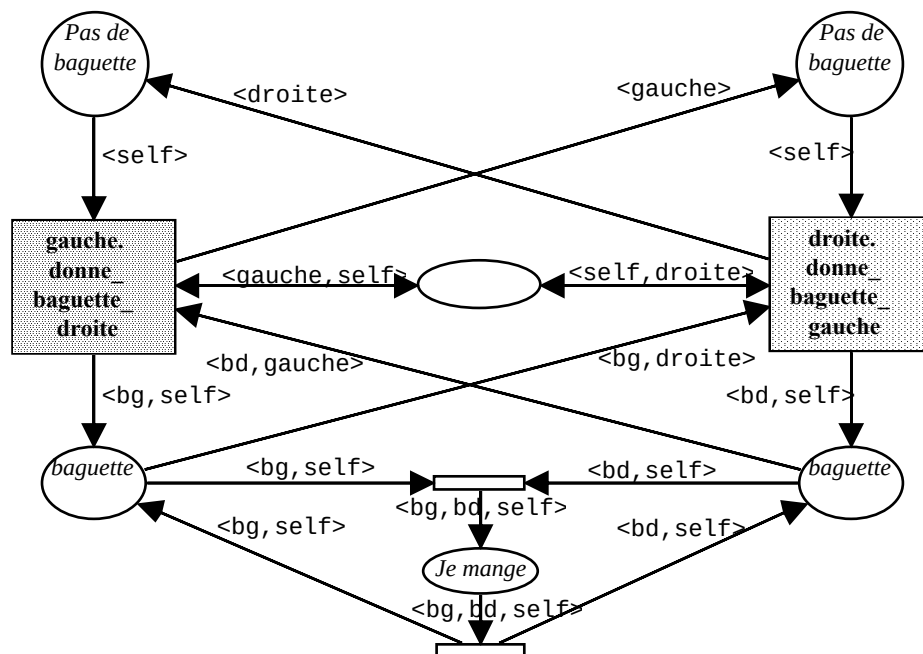


Figure V.21 - Réseau simplifié de la table de philosophes

Le système décrit ci-dessus est un système entièrement statique, ou le nombre d'instances n'évolue pas. Il nous apparaît que c'est pour de tels systèmes que la modélisation par objets composites est la plus adaptée.

7. CALCUL DES PROCESSUS SEQUENTIELS

Plusieurs méthodes de conception de systèmes concurrents [Ward...85], [Harel...88] débutent le processus de modélisation par l'identification des processus séquentiels du système. Cette identification peut se révéler subjective ou difficile à réaliser lors d'une étape si précoce.

Dans le WSCS, les processus séquentiels du système peuvent être identifiés en cherchant des invariants de place [Colom...86]. Ces processus séquentiels ne sont plus limités par les frontières entre objets, puisqu'un même processus peut être distribué sur plusieurs objets. Nous avons identifiés de tels processus séquentiels dans l'exemple des philosophes (§V.6).

L'identification des processus séquentiels pourrait être utilisée pour générer des squelettes de codes dans un langage parallèle tel que Ada.

C'est un des avantages du formalisme des Objets Coopératifs que de pouvoir ignorer la notion de processus séquentiel pendant toute la conception préliminaire, et d'obtenir cette décomposition par analyse de la conception produite.

8. CONCLUSION SUR LE RESEAU GLOBAL

La complexité liée à l'existence du polymorphisme, à la création dynamique d'objets et à la topologie dynamique de la relation d'utilisation est traduite par une complexité accrue de la structure du réseau et de son marquage.

Tout le processus de construction du WSCS repose sur deux mécanismes de base :

- L'unification, qui permet de retrouver, dans le marquage de l'ObCS d'extension d'une classe, les jetons qui caractérisent l'état d'une instance ;
- La génération d'identificateurs, utilisée aussi bien pour la génération de nouveaux noms d'objet que pour la génération d'identificateurs d'appel.

Le mécanisme de liaison tardive, généralement implanté dans les langages à objet par des chaînes de pointeurs vers des descripteurs de classe, est implanté ici par l'unification, opération puissante mais relativement simple à mettre en œuvre et informatiquement bien maîtrisée.

Une fois le WSCS construit, le système est représenté par un réseau prédicat/transition, où toute la structure de données et de contrôle des objets est mise à plat. L'évolution du système est alors complètement définie par :

- La structure du WSCS,
- Le code des opérations internes,
- Le marquage initial du WSCS.

Si on voulait obtenir un degré de formalisation supérieur, il faudrait soit se dispenser totalement d'opérations internes algorithmiques (le lecteur pourra vérifier qu'elles sont très peu utilisées dans ce mémoire), soit s'astreindre à les exprimer elles-mêmes par un RdP. L'évolution du système ne dépendrait alors plus que de la structure et du marquage du WSCS et de la sémantique des opérateurs prédéfinis sur les types prédéfinis (+, -, *, ...).

On peut faire une analogie entre le WSCS d'un système d'Objets Coopératifs et la liste en langage d'assemblage qui résulte de la compilation d'un programme informatique en langage de haut niveau : comme cette liste, le WSCS est construit par un processus automatique, et sa structure présente assez peu d'intérêt pour le concepteur, qui ne s'y reportera que s'il constate que son modèle fonctionne de manière pathologique, afin d'effectuer des opérations de "débogage" de bas niveau. Toutefois c'est le WSCS, tout comme la traduction en langage d'assemblage d'un programme de haut niveau, qui définit "in fine" la sémantique du modèle, et qui assure son exécutabilité.

Chapitre VI. Extension du formalisme

Nous avons présenté au chapitre III les Objets Coopératifs, en nous limitant volontairement au "noyau dur" du formalisme, c'est à dire aux concepts qui sont à la base de notre approche, et qui nous paraissent suffisants pour aborder la modélisation d'une classe assez large de systèmes concurrents.

L'objet de ce chapitre est de présenter un certain nombre d'extensions possibles à ce noyau de base, afin de faciliter la modélisation dans des domaines plus ciblés, ou d'émuler facilement des constructions ou des primitives rencontrées dans d'autres approches :

- Nous proposons tout d'abord deux autres **modes d'invocations** des services, qui permettent d'avoir des protocoles de communication entre objets différents du rendez-vous (qui est un appel de type question / réponse) ;
- Nous proposons ensuite d'utiliser le mécanisme des **places triées** (§II.2.7) pour définir des **priorités entre invocations** d'un service ;
- Nous introduisons enfin la notion de **flot de donnée** entre objets, c'est à dire une communication qui n'induit pas de flot de contrôle. Cette notion est ensuite étendue pour définir les **objets passifs**, objets dont tous les services sont des flots de données.

1. MODES D'INVOCATION DES SERVICES

Le rendez-vous, défini au §III.4.6, est le mode de communication élémentaire entre objets. Un client qui invoque un service dans une de ses TI doit s'attendre à subir un certain délai pour le franchissement de cette transition, correspondant au temps mis par le serveur pour accomplir le service.

Ce délai dépend uniquement de l'état du serveur, et peut éventuellement devenir infini si le marquage de l'ObCS du serveur est tel que le service ne puisse plus devenir accessible. Dans ce cas, l'invocation ne se termine jamais, et le franchissement correspondant reste pour toujours suspendu chez le client.

Afin de donner au client plus de contrôle sur ses invocations, nous allons définir deux nouveaux **modes d'invocation**, en les munissant d'une représentation graphique, en décrivant, comme pour le rendez-vous, leur sémantique formelle par le RPO équivalent à la séquence d'appel et en explicitant la manière de les intégrer dans le procédé de construction du WSCS décrit en §V.3. Il faut souligner que ce sera toujours au client de décider du mode d'invocation d'un service : le serveur n'est pas informé du mode avec lequel il est invoqué.

1.1. Invocation non confirmée

Dans certains cas, il peut être significatif pour un client d'envoyer à un serveur un **signal** ou un message, sans attendre en retour ni résultat ni accusé de réception. Ce mode de communication est utile par exemple pour décrire des situations de diffusion d'information (multi-casting), ou pour implémenter le mécanisme de la **continuation** des langages d'acteur [Hewitt 77].

Un telle invocation sera dite **non confirmée** et représentée graphiquement par un rectangle surmontant un trait (Figure VI.1). Dans une invocation non confirmée, les valeurs de retour du service invoqué sont bien entendu inaccessibles, et ne peuvent être affectées à des variables de sortie de la transition.

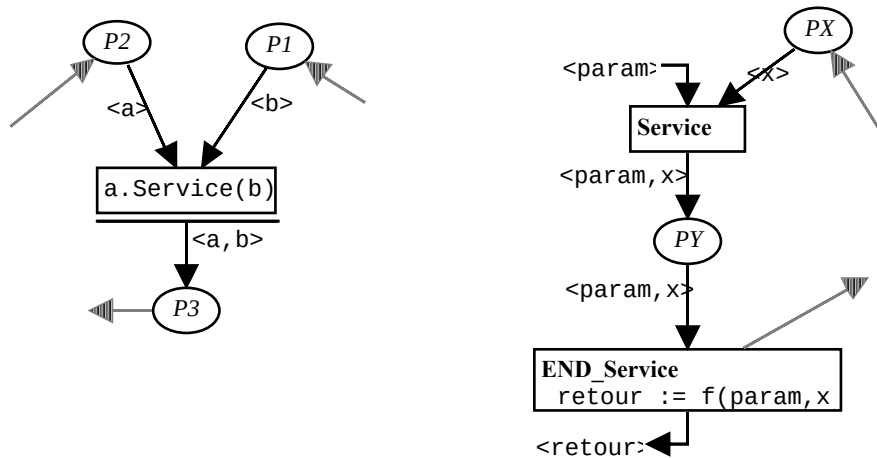


Figure VI.1 - Syntaxe graphique d'une invocation non confirmée

L'implémentation d'un tel mode de communication est très simple : lors de la première étape de construction du WSCS (§V.3.1) les places de sortie de la TI sont reliées à la transition d'appel et non plus à la transition de complétion ; le reste du procédé de construction demeure identique. La transition de complétion, notamment, a alors pour but de consommer le jeton typé résultat de l'exécution du service.

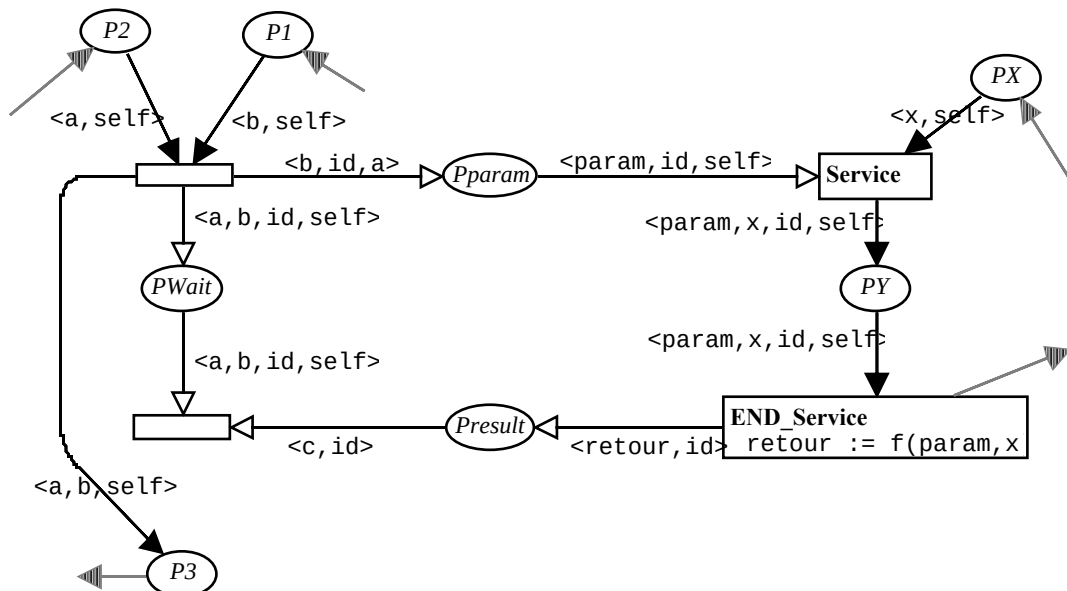


Figure VI.2 - Sémantique d'une invocation non confirmée

La figure VI.2 illustre la syntaxe graphique d'une invocation non confirmée, et la figure II.40 en donne la sémantique, avec une partie du WSCS généré par une telle invocation.

Une invocation non confirmée est l'équivalent d'un type de message "PAST" du langage ABCL [Yonezawa...86] (§I.2.3), ou d'une requête ASER de la méthode HOOD (§I.3).

1.2. Invocation gardée

Les mécanismes de temporisation des RPO décrits en §II.2.8 sont utilisables lors de la définition des ObCS. On peut alors décrire pour un objet non seulement les séquences légales d'invocation de services et les contraintes sur leurs synchronisations, mais aussi les aspects liés à leurs dates ou durées d'exécution. Dans la suite, nous utiliserons uniquement la notation des RPO à arcs temporels, aussi bien pour décrire cette extension que dans l'étude de cas du §VII.1.

Lorsque l'on introduit des temporisations dans la description de l'ObCS d'un serveur, il devient significatif pour un client de se préoccuper du délai de prise en compte d'une invocation. Un client peut notamment refuser d'attendre longtemps qu'un serveur soit prêt à répondre à son invocation. Il peut alors associer une **garde** (un certain nombre d'unités de temps) à l'invocation, exprimant ainsi qu'il accepte d'attendre cette durée avant que sa requête soit prise en compte par le serveur. Ce mécanisme est comparable à l'attente gardée du langage Ada. La sémantique intuitive d'une invocation gardée est la suivante :

- Les jetons sont retirés des places d'entrées de la TI, et les paramètres correspondants sont placés dans la place d'attente. Deux choses peuvent alors se produire :
 - La transition d'accord devient franchissable chez le serveur avant l'expiration du délai de garde. Dans ce cas, le serveur exécute le service comme dans un simple rendez-vous ;
 - Le délai de garde expire avant que la transition d'accord ne devienne franchissable. L'invocation est alors annulée, et la variable locale *timeout* est mise à VRAI. Les résultats de l'invocation sont alors indéfinis. Une règle d'émission (cf §II.2.3) est toujours associée aux invocations par rendez-vous gardé, afin de définir quelles sont les actions à effectuer par le client en cas de timeout (Figure VI.3).

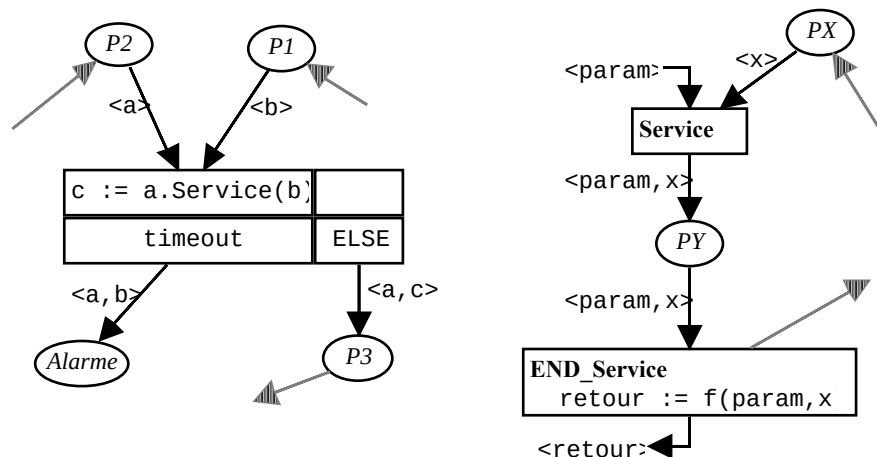


Figure VI.3 - Syntaxe d'une invocation gardée

Ce type d'invocation doit être pris en compte lors des étapes 5 et 6 de la construction du WSCS :

- Après l'étape 5, il existe pour chaque service une seule place paramètre et une seule place résultat, provenant de la fusion des places paramètres et résultat pour toute la hiérarchie d'héritage. Si il existe dans le système une invocation gardée de ce service, on connecte la place

paramètre et la place résultat par l'intermédiaire d'une **transition d'annulation**, dont l'occurrence a pour effet de "court-circuiter" le traitement normal associé au service.

- Le type de la place paramètre est étendu par l'adjonction d'un élément dénotant le délai associé à l'invocation. On utilise, pour décrire la temporisation, la notation des RdP à arcs temporels [Bastide...89] (cf §II.2.8). Les arcs connectant la place paramètre et les transitions d'accord sont temporisés par l'expression $[0, \text{delay}]$ ou delay dénote la valeur de l'élément correspondant dans la place paramètre. L'arc qui connecte la place paramètre à la transition d'annulation est temporisé par l'expression $]\text{delay}, \bullet[$;
- Le type de la place résultat est étendu par l'adjonction d'un élément booléen, dénotant l'occurrence d'un timeout lors de l'invocation du service. Les étiquettes des arcs reliant les transitions de retour du service à la place résultat sont étendues pour initialiser cet élément à FAUX, alors que l'arc en provenance de la transition l'initialise à VRAI ;

- Lors de l'étape 6, les étiquettes des arcs connectant les transitions d'invocation avec la place paramètre sont étendues afin de transmettre la valeur du délai de garde avec les autres paramètres de l'invocation. Pour les invocations par rendez-vous non gardé, la valeur infinie (∞) est transmise.

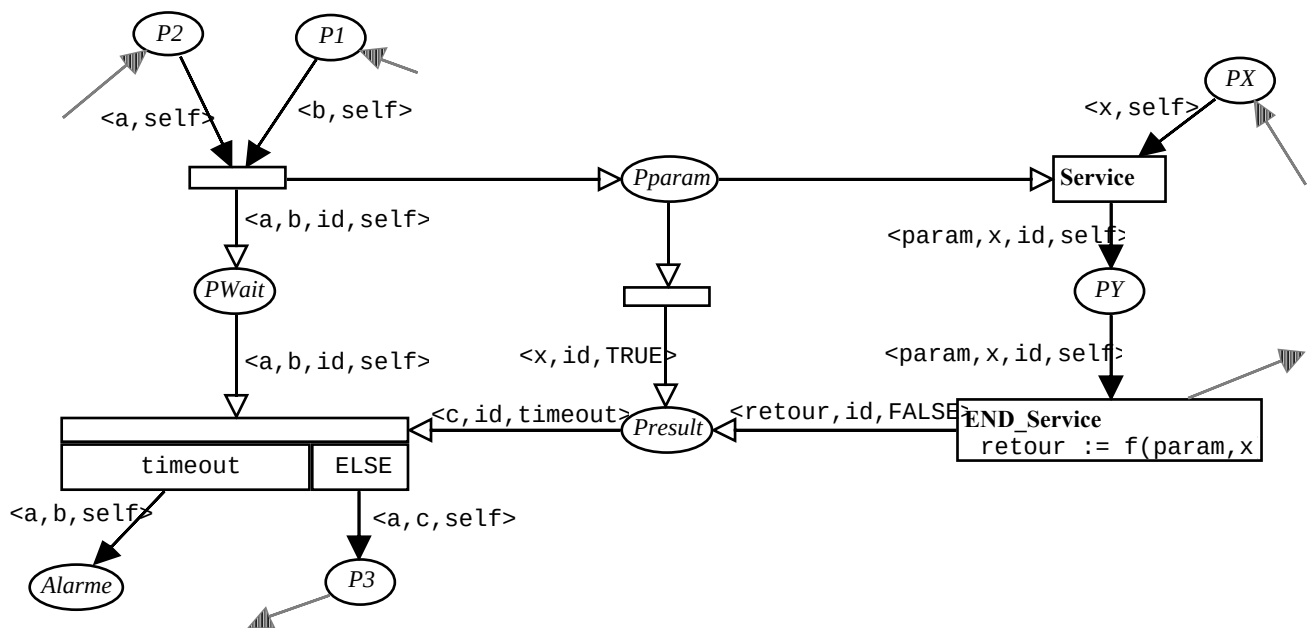


Figure VI.4 - Sémantique d'une invocation gardée

La syntaxe graphique d'une invocation gardée est illustrée en figure VI.3, et sa sémantique en figure VI.4, avec une partie du WSCS généré par une telle invocation. L'invocation gardée est l'équivalent d'une TOER (Timed Out Execution Request) de la méthode HOOD.

1.3. Communication synchrone

Le mode d'invocation standard d'un service est le rendez-vous, qui correspond aux requêtes HSER (Highly Synchronous Execution Request) de la méthode HOOD, établissant une communication de type question/réponse).

L'invocation non confirmée correspond aux requêtes ASER (ASynchronous Execution Request), et l'invocation gardée permet de définir des requêtes TOER (Timed Out Execution Request).

Malgré nos efforts, nous ne sommes pas parvenus à trouver une construction simple permettant d'émuler dans tous les cas les requêtes LSER (Loosely Synchronous Execution Request), où l'appelant est bloqué jusqu'à la prise en compte de son appel, et peut ensuite continuer son exécution.

Intuitivement, la construction qui émulerait ce mode de communication consisterait en une fusion de la **transition d'appel** du client avec la **transition d'accord** du serveur ; malheureusement cette solution est impossible, car le même service peut être invoqué par plusieurs transitions différentes dans l'ObCS du client, et surtout par un nombre indéterminé de clients de classes différentes. Le problème est encore compliqué par le **polymorphisme**, car la classe de l'objet qui reçoit l'appel peut varier suivant la hiérarchie d'héritage.

Pour pouvoir émuler une requête LSER il faudrait donc que le service concerné soit invoqué dans une seule transition du système, et que l'entité qui dénote le serveur soit une constante, ou soit d'une classe feuille de la hiérarchie d'héritage.

2. ORDONNANCEMENT DES INVOCATIONS

Au cours de l'activité d'un système, il peut fort bien arriver qu'un objet reçoive plusieurs invocations concernant le même service avant d'être en mesure d'en servir une. Un objet peut avoir plusieurs invocations en attente pour un des services qu'il offre, ce qui se matérialise par plusieurs jetons dans la place paramètres de ce service.

En l'absence de toute précision, l'ordre dans lequel ces invocations seront servies (c'est à dire l'ordre dans lequel les jetons de la place paramètres seront consommés par occurrence de la transition d'accord) n'est pas déterminé. Ceci est à contraster avec la sémantique du rendez-vous dans le langage Ada, où les appels en attente sur une entrée de tâche sont servis dans l'ordre FIFO.

Le modélisateur peut souhaiter préciser l'ordre de traitement des invocations, réduisant de la sorte l'indéterminisme du comportement de l'objet qu'il décrit. La spécification d'un ordre de traitement des requêtes peut parfois se révéler nécessaire afin d'éviter des phénomènes de privation, où la requête déposée par un client n'est jamais servie bien que l'objet serveur traite effectivement la requête pour d'autres clients, et que la sienne soit recevable (c'est à dire qu'elle satisfasse la précondition de la transition d'accord).

Les mécanismes d'**ordonnement des marquages** décrits au §II.2.7 sont exactement adaptés à l'ordonnement des invocations : il suffit pour cela d'affecter une fonction de tri à la place paramètre du service sur lequel on souhaite définir une priorité. Le mécanisme de tri le plus souvent utilisé sera celui de la File (FIFO), dans lequel les exécutions de service ont lieu dans l'ordre des invocations. Toutefois, toutes les primitives de tri décrites au §II.2.7 sont applicables aux places paramètres ; on pourra ainsi, par exemple, spécifier que les requêtes pour un service doivent être traitées dans l'ordre induit par la valeur d'un de ses paramètres, et permettre ainsi à un serveur de définir des priorités entre ses différents clients, par exemple.

Syntaxiquement, l'ordre de traitement des invocations d'un service sera déclaré lors de la spécification de celui-ci, en utilisant les mêmes mots-clés que pour spécifier l'ordonnement d'une place. La figure VI.5 illustre quelques spécifications possibles de priorités sur les invocations, en prenant l'exemple du travail quotidien d'une secrétaire.

Class Specification Secrétaire Services Réponds_au_téléphone Hard FIFO ; -- Les appels téléphoniques sont mis en attente, et -- traités dans l'ordre d'arrivée. Envoie_règlement<f:Fournisseur,m:Montant> Soft Descending m ; -- Les règlements sont transmis aux fournisseurs en -- donnant la priorité aux montants les plus élevés. Frappe_le_cours<e: Enseignant,b : Brouillon>:<f: Fichier_word4>; -- L'ordre n'est pas spécifié : la secrétaire frappe les -- cours avec une priorité connue d'elle seule.
--

Figure VI.5 - Exemples de la syntaxe de services ordonnancés

Il nous faut maintenant décrire la manière dont l'ordonnement des invocations d'un service s'intègre dans le procédé de reconstruction du WSCS décrit en §V.3.

- Comme on l'a défini en §II.2.7, les places faiblement triées (mention **Soft**) sont une extension au formalisme des RPO : ce mécanisme étend explicitement le pouvoir d'expression du formalisme, et il n'est pas possible de trouver un motif équivalent qui se restreigne à la définition de base des RPO. La définition d'un service ordonnancé de manière faible n'a donc pas d'influence sur le procédé de construction du WSCS, mis à part le fait que la place paramètre de ce service doit être faiblement triée;
- Les places fortement triées (mention **Hard**), au contraire, sont implémentées comme une abréviation des RPO, et une telle place génère donc un motif particulier qui en tient lieu. La définition d'un service ordonnancé de manière forte doit donc être prise en compte lors de la construction du WSCS. Le motif correspondant au critère de tri de la place doit être inséré lors des étapes 1 et 2 du procédé, au moment où sont introduites les places paramètres ;
- L'ordre porte toujours sur la **prise en compte** de l'invocation par le serveur, et non pas sur la fourniture des résultats. Toutefois, l'OpCS d'un service peut être défini de telle sorte que l'ordre de fourniture des résultats soit le même que celui de la prise en compte.

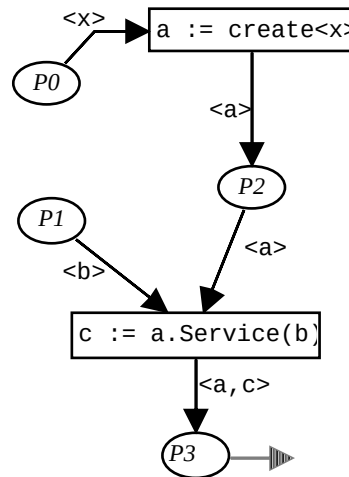
Class Client

ObCS

```

P0, P1 : <i : INTEGER>;
P2 : <a : Serveur>;
P3 : <a : Serveur, c : Integer>

```



Class Serveur

Services

S1<param : INTEGER> : <retour : INTEGER> Hard L

ObCS

```

PX : <x : INTEGER>;
PY : <x, y : INTEGER>;

```

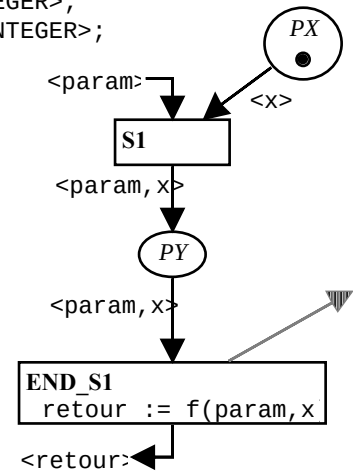


Figure VI.6 - Invocation d'un service ordonné

A titre d'exemple, nous illustrons comment une TI référençant un service géré en pile (LIFO) doit être étendue par le procédé de construction du WSCS. On peut vérifier que le service est bien géré en pile **au niveau de chaque instance** de la classe serveur, et non pas au niveau de la classe elle-même. L'ordre de traitement entre les différentes instances de la même classe demeure indéterministe.

La figure VI.6 montre des extraits des ObCS du client et du serveur. La figure VI.7 montre le WSCS généré par les extraits correspondants. Il faut remarquer comment le marquage de la place Sommet est initialisé à chaque instanciation de la classe Serveur, par occurrence de la transition *Create*.

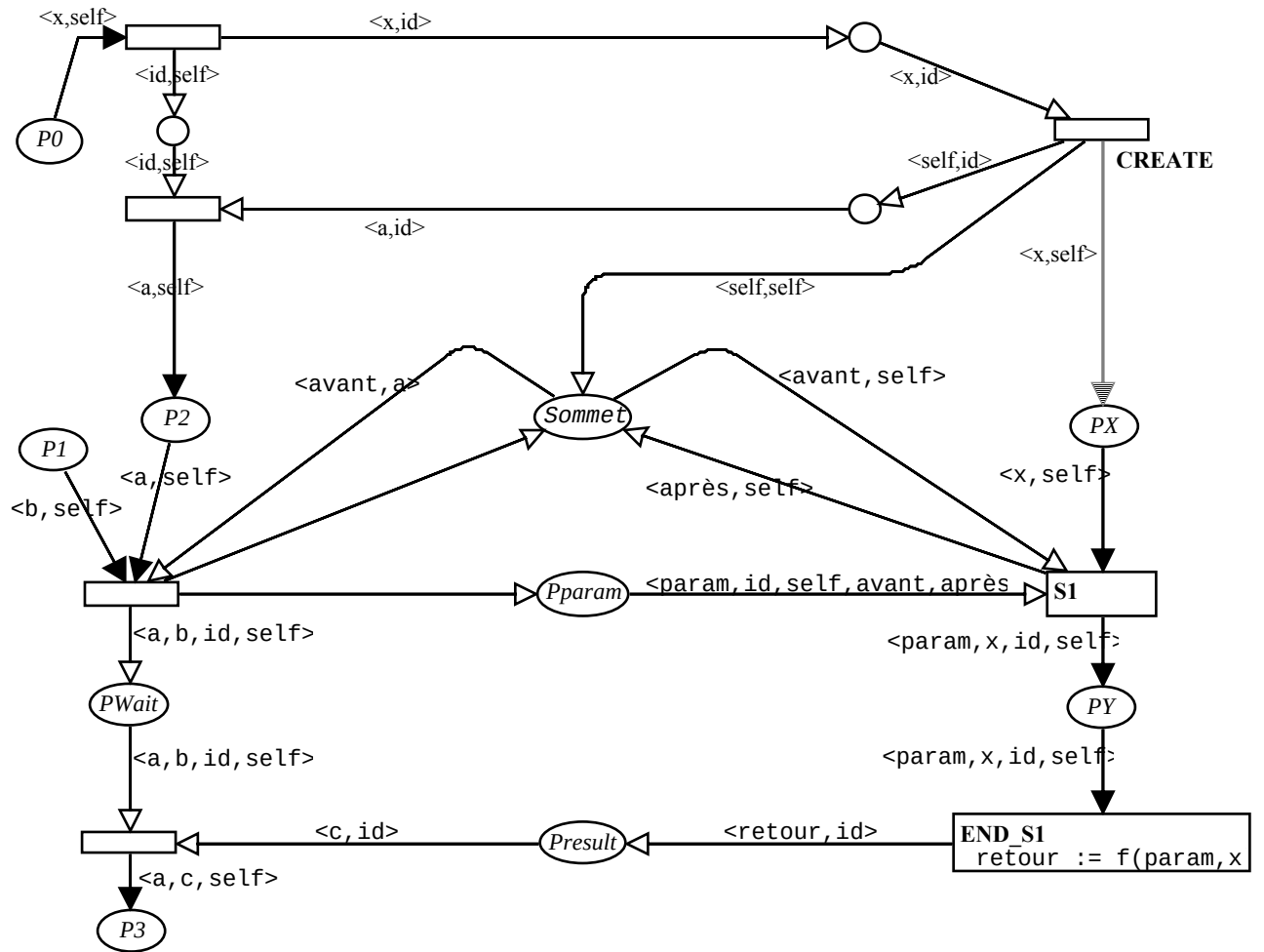


Figure VI.7 - WSCS d'un service LIFO

3. FLOTS DE DONNEES

En général, une invocation a pour conséquence l'initiation, la terminaison, l'interruption ou la réactivation de **flots de contrôle** internes à l'objet serveur. Dans de nombreux cas, cette action sur la structure de contrôle du serveur est triviale, voire inexistante ; nous avons déjà rencontré un tel cas : il s'agit des **services de mise à jour** des attributs d'un serveur (cf §III.3), qui offrent une manière standardisée de modifier la valeur des attributs publics d'un objet. Dans les ObCS (de spécification ou d'implémentation), un service de mise à jour apparaît comme une transition non connexe au reste du réseau : la connexité sera restaurée au moment de la construction de l'ObCS d'extension de la classe, par l'introduction des places-attributs¹² (cf §V.3.3). La spécification et la sémantique d'un tel service sont rappelées en figure VI.8.

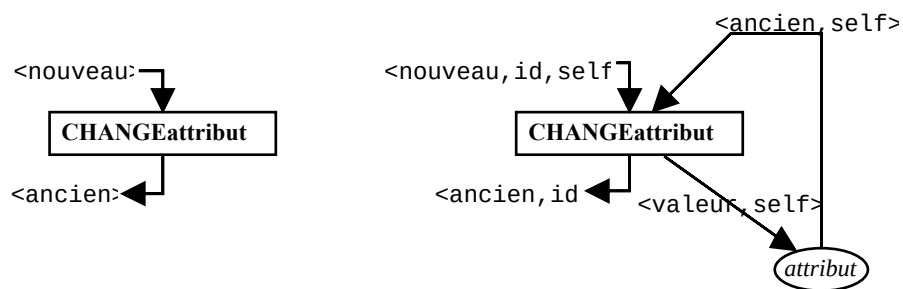


Figure VI.8 - Spécification et sémantique d'un service de mise à jour

Ces services ont des propriétés remarquables :

- Une invocation les concernant n'a aucune influence sur le contrôle interne du **serveur** : la répartition du marquage dans son ObCS n'est pas modifiée, ni la valeur de ce marquage, si l'on excepte celui de la place-attribut concernée par la modification. D'autre part l'exécution de ce service est atomique (elle correspond au franchissement de la transition de service) ;
- Une telle invocation n'a pas non plus d'influence sur le contrôle du **client** de l'appel : ce service est toujours disponible (pour peu que l'attribut concerné ait été initialisé), et son exécution est également atomique. Du point de vue du client, l'exécution du service n'est donc pas différente de l'exécution d'une opération interne algorithmique.

Ces deux propriétés sont les caractéristiques d'un **flot de données** entre les objets. Dans le cas d'un service de mise à jour, ce flot de données est bidirectionnel : la nouvelle valeur de l'attribut est transmise du client vers le serveur, et l'ancienne est renvoyée du serveur au client. L'accès en lecture à un attribut public (§III.3) ou son initialisation constituent un flot de données mono-directionnel.

¹²Dans le cas où l'ObCS d'extension serait non-connexe, on serait en droit de se demander si le modélisateur a bien identifié un **objet**, c'est à dire une entité fortement cohérente et faiblement couplée.

3.1. Définition

Nous proposons donc de généraliser cette notion de flots de données entre objets, en caractérisant une nouvelle classe de services appelés services **data-flows** :

Définition VI.I - Services Data-Flow

Un **Data-Flow** est un service :

- Associé à une transition non connexe au reste de l'ObCS de spécification (et donc défini comme étant toujours disponible) ;
- Pour lequel l'action de la transition associée dans l'ObCS d'implémentation contient : soit un appel d'opération interne algorithmique soit une invocation concernant un autre data-flow.

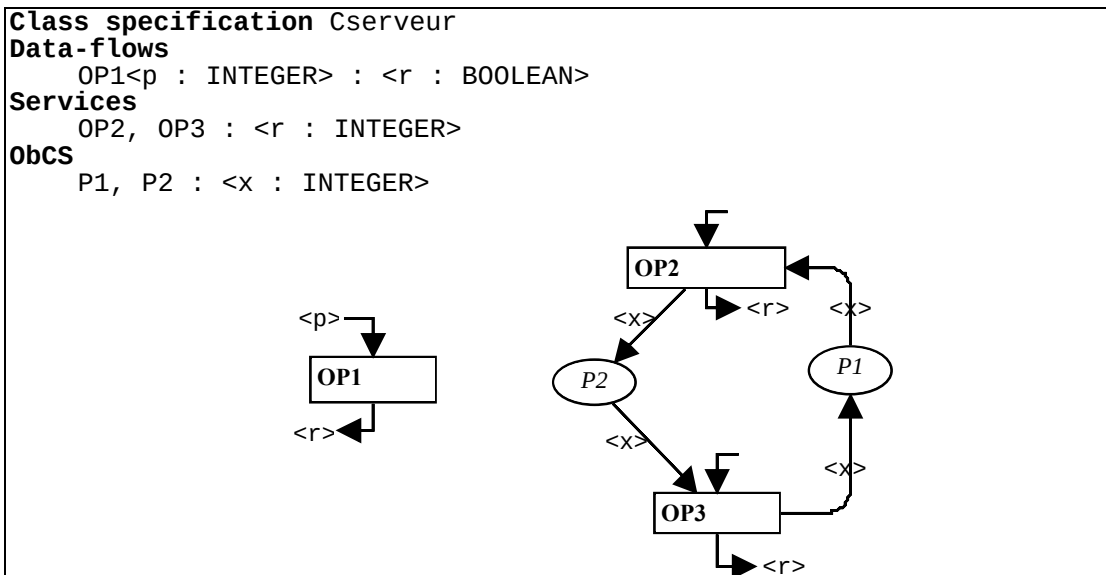


Figure VI.9 - Classe *Cserveur*, publiant un data-flow

Les data-flows sont décrits dans une section spéciale de la spécification d'une classe, avec la syntaxe habituelle des services (figure II.47). On peut se dispenser de faire figurer la transition associée (non connexe) dans l'ObCS de spécification. De même, dans l'ObCS d'implémentation, on ne montrera pas la transition associée : par convention, il suffit de fournir une opération interne de même nom que le service ; c'est cette opération qui doit être exécutée en cas d'invocation.

La figure VI.9 illustre une spécification d'objet présentant un data-flow. On peut se dispenser de faire figurer dans l'ObCS la transition associée à *OP1*.

Il faut préciser comment les data-flows sont pris en compte dans le procédé de construction du WSCS. La figure VI.10 montre comment l'ObCS d'extension sera généré en suivant le procédé du §V.3. On

voit que l'exécution du data-flow n'a pas d'influence sur le contrôle du serveur. La transition associée au service *OP1* exécute l'opération interne *OP1*, qui renvoie un résultat fonction du paramètre reçu et de la valeur de deux attributs.

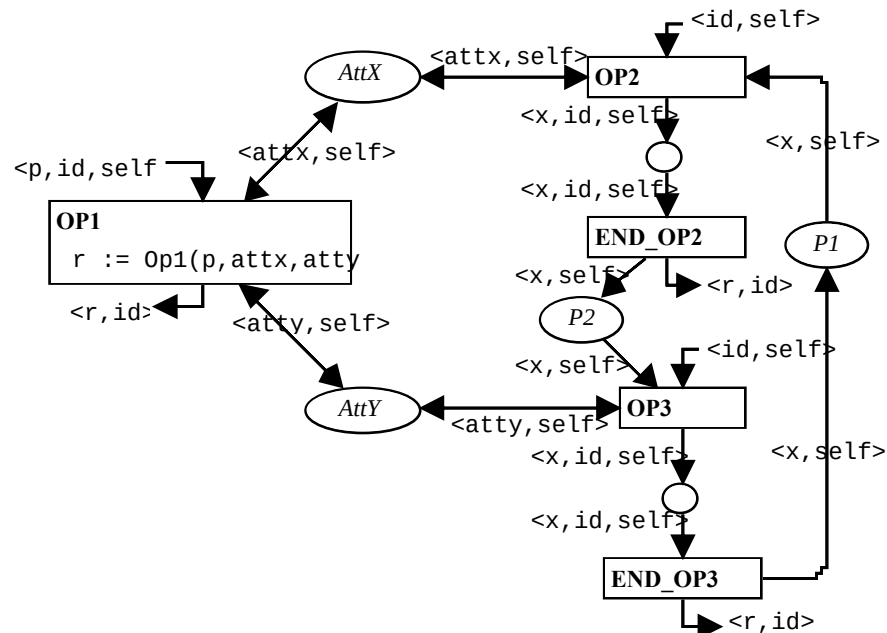


Figure VI.10 - ObCS d'extension de *Cserveur*

3.2. Intégration dans le réseau global

Il nous faut cependant modifier le procédé de construction du WSCS, de manière à ce que l'exécution du data-flow puisse être également considérée comme le franchissement d'une transition unique chez le client. Ceci est réalisé par un procédé similaire au traitement des TA (transitions d'accès) : la TI qui fait appel au data-flow est connectée en entrée et en sortie aux places-attributs de l'objet serveur qui interviennent dans l'opération interne associée, et l'appel à l'opération est intégré à la transition d'invocation (figure VI.11). L'opération interne correspondante (Op1) doit accéder au même ensemble d'attributs le long de la hiérarchie d'héritage, afin que les places-attributs puissent être fusionnées.

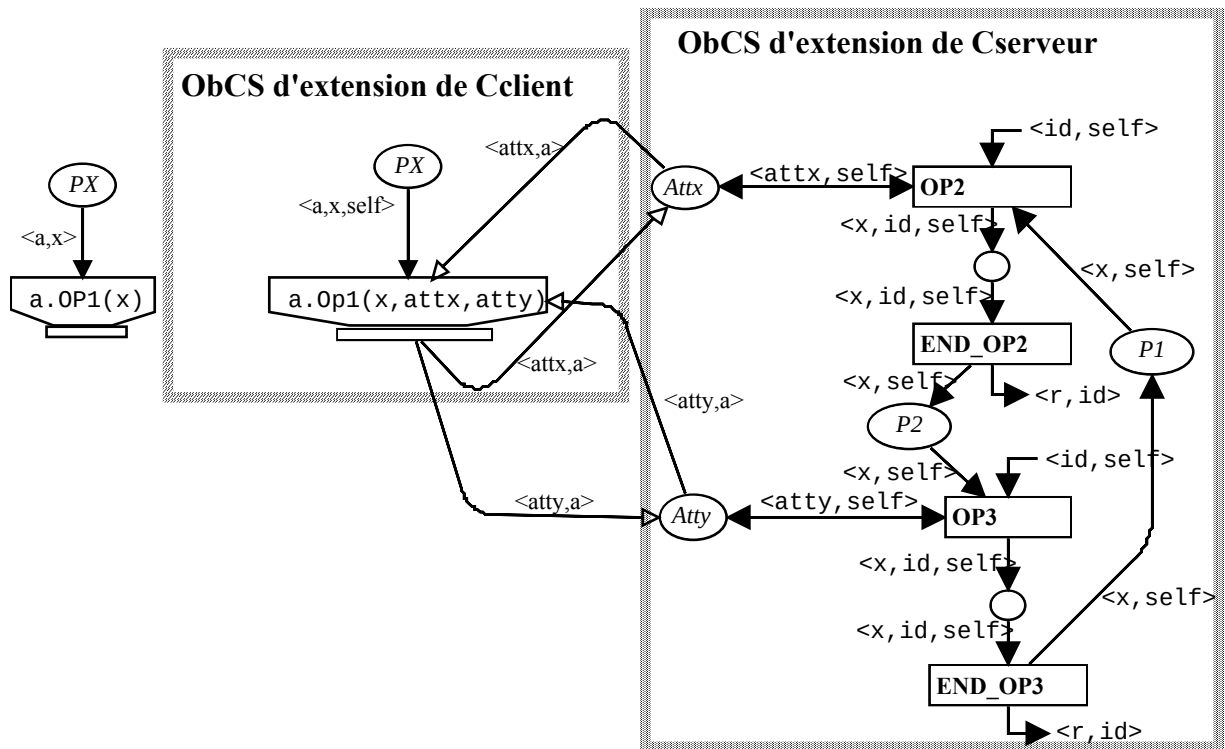


Figure VI.11 - WSCS généré par l'invocation d'un data-flow

Une invocation concernant un data-flow n'a aucune influence sur le contrôle du client ni sur celui du serveur. Une telle invocation peut donc intervenir dans une précondition, ou dans le corps d'une opération interne algorithmique du client.

Ceci permet de remédier aux limitations des opérations internes, décrites en §III.4.2 : la raison pour laquelle on avait interdit l'utilisation des invocations dans le code des opérations internes, était d'imposer que tous les flots de contrôle entre objets soient définis dans les ObCS d'implémentation. Un data-flow ne représentant pas un flot de contrôle, cette restriction peut être levée sans remettre en cause la sémantique formelle de la coopération entre objets décrite par le WSCS.

Dans ce mode de communication, c'est le client qui exécute le code algorithmique associé à un service. C'est la sémantique que la méthode HOOD donne aux opérations **sans contrainte fonctionnelle d'activation** (cf §I.3).

L'expression **a.OP1<x>**, qui apparaît dans la précondition, doit être interprétée comme un appel de méthode dans un langage à objets séquentiel, avec liaison tardive : la variable **a** peut être liée à des objets dont la classe est descendante de la sienne propre. Il faut donc rechercher dans le graphe d'héritage l'implémentation correcte de l'opération **OP1**.

On peut donner une manière intuitive de considérer les data-flows : en exportant un data-flow, une classe publie le code algorithmique qui permet d'accéder à ses attributs, et non pas les attributs eux-mêmes. On réalise ainsi une forme d'encapsulation qui n'existe pas si l'on publie directement les attributs.

Un data-flow ne peut pas être considéré exactement comme un attribut calculé, puisque la valeur qu'il renvoie peut dépendre d'un paramètre reçu ; un attribut calculé peut être considéré comme un cas simple de data-flow, sans paramètre et donc unidirectionnel.

4. OBJETS PASSIFS

Le formalisme des Objets Coopératifs, tel qu'on vient de le définir, est bien adapté pour décrire des entités de haut niveau, des **agents** actifs dans l'évolution du système, qui ont leur comportement propre et qui peuvent être à l'origine de flots de contrôle et non pas seulement les subir. Un Objet Coopératif est le plus souvent identifiable à un **processus** (non séquentiel) du système dans lequel il intervient.

Toutefois, un système est le plus souvent décrit à la fois par les processus qu'il contient et les **données** que ces processus manipulent. Une donnée d'un système est caractérisée par son caractère **passif** vis-à-vis des flots de contrôle qui la manipulent : elle n'est à l'origine d'aucun flot de contrôle, elle ne possède aucun flot de contrôle interne, et elle n'impose aucune contrainte fonctionnelle d'activation sur les services qu'elle offre.

A la lumière de la discussion de la section précédente, nous pouvons définir comment une entité de ce type, que nous nommerons **objet passif**, est caractérisée dans le formalisme des Objets Coopératifs :

Définition VI.2 - Classe d'Objets Passifs

On appelle **classe d'objets passifs** une classe d'Objets Coopératifs dont tous les services publiés sont des Data-Flows.

Une instance d'une telle classe sera appelée **objet passif**.

La façon dont cette définition caractérise les objets passifs est voisine de celle de la méthode HOOD (§ I.3), avec quelques différences toutefois :

- HOOD précise que les objets passifs n'ont pas d'ObCS ; dans notre cas, ils serait plus correct de dire que l'ObCS de spécification des objets passifs est trivial (non connexe, car constitué exclusivement de transitions de services isolées) et que leur ObCS d'implémentation consiste uniquement à assurer l'exclusion mutuelle entre les services. L'avantage de cette approche est que la sémantique d'un objet passif utilisé par plus d'un objet actif est précisément définie, ce qui n'est pas le cas dans HOOD ;
- HOOD interdit à un objet passif d'utiliser un objet actif ; notre définition est moins restrictive, car elle permet à un objet passif d'utiliser les data-flows publiés par une classe d'objets actifs.

Il faut remarquer, d'après la définition VI.1 (Data-Flows) que toutes les opérations offertes par un objet passif sont exécutées par le client ; un objet passif n'a donc pas de "processeur" dédié, si on prend la métaphore d'un processus informatique, ni "d'énergie vitale", avec une métaphore biologique [Sibertin 91]. Un objet passif n'est donc pas autonome, et il est nécessaire qu'un client actif l'invoque et exécute lui-même les opérations qu'il offre.

Cette définition a pour avantage d'éviter la distinction nette entre données et processus imposée par d'autres approches. On peut alors qualifier l'aspect "actif", "vivant" ou "autonome" d'un objet suivant une échelle très fine ; en descendant cette échelle de l'actif vers le passif, on trouverait :

- Des classes qui ne publient que des services (pas d'attributs ni de data-flows) et qui ont une activité spontanée décrite par leur ObCS. Ce sont les **processus** (pas forcément séquentiels) du système, et ils sont très faiblement couplés à leur environnement ;
- Des classes qui ne publient que des services, et qui n'ont pas d'activité spontanée. On peut les comparer à des **moniteurs** (§ II.2.1), qui encapsulent une donnée et les mécanismes d'accès concurrents à cette donnée ;
- Des classes qui publient services, data-flows et/ou attributs. Ce sont des intermédiaires entre des données et des processus. Leur couplage avec l'environnement est plus fort, du fait de l'implantation par places partagées des attributs et des data-flows ;
- Enfin les objets passifs, qui n'offrent pas de services, mais uniquement des data-flows et/ou des attributs. Ce sont les **données** du système.

PARTIE III :

ETUDES DE CAS ET EXECUTION DES MODELES D'OBJETS COOPERATIFS

Chapitre VII. Etudes de cas

Nous présentons dans ce chapitre deux études de cas traitées par les Objets Coopératifs. Nous avons choisi ces études dans deux domaines très différents, dans le but de démontrer l'étendue du domaine d'application de notre formalisme.

Ces deux études ne sont pas des "cas d'école", mais au contraire des exemples réels et significatifs de leur domaine :

- La première, qui consiste en la modélisation d'un atelier flexible de production, est tirée d'un cahier des charges émis par une société privée dans le but de choisir un outil de simulation [Fritschy 89] ;
- La deuxième, une interface personne/logiciel, est extraite du dossier de spécification d'un logiciel d'ordonnancement court terme d'atelier, destiné à l'industrie aéronautique, et qui a été réellement implémentée (par des techniques de programmation conventionnelles toutefois).

1. MODELISATION D'UN ATELIER FLEXIBLE

1.1. Présentation générale de l'atelier

L'atelier proposé est basé sur une architecture modulaire dite en "marguerite". La fabrication est multi-produits (3 produits : *PA*, *PB*, et *PC*). Ces produits circulent sur des palettes. Ces palettes portent un code qui détermine les opérations que doit subir le produit. Il est donc possible d'affecter à chacune d'elles une gamme de fabrication différente qui permet de définir la circulation des palettes au sein de l'atelier, ainsi que les opérations à effectuer sur chaque poste de travail.

Le plan de l'atelier est montré Figure VII.1. Il se compose de 3 éléments :

- un îlot d'assemblage (*Ilot1*) pour la fabrication complète du produit *PA* et pour la fabrication partielle de *PB* ;
- un îlot d'assemblage (*Ilot2*) pour finir la fabrication du produit *PB*, et pour la réalisation complète du produit *PC* ;
- un système de convoyage, appelé *Distributeur*, qui d'une part assure la distribution des palettes vers les îlots, et qui d'autre part joue un rôle de passerelle Ilot1 - Ilot 2. Il est constitué de modules de transfert (à 1 seule voie) et de module de dérivation vers les îlots.

Chaque îlot est constitué de 5 modules de production comportant chacun un poste de travail robotisé multi-opérations (Figure VII.1), et de 3 modules de transports reliés entre eux. Ces modules se composent de 2 circulateurs en anneau (ou pistes), avec des dérivations avant et après chaque poste.

Les postes sont situés sur la piste extérieure.

Le fonctionnement des machines est asynchrone.

Une palette tournant dans un îlot et concernée par un poste de travail passe de l'anneau intérieur à l'anneau extérieur via les dérives prévues à cet effet. Après l'arrêt devant un poste par deux bloqueurs, la palette est indexée pour assurer un positionnement précis, puis l'opération a lieu. Lorsqu'elle est terminée, la palette peut aller au poste suivant soit par l'anneau extérieur, soit par l'anneau intérieur (Figure VII.2).

D'un point de vue mécanique, le transport des produits est assuré par des tapis roulants. En cas de blocage des palettes (aux stoppeurs), il y a glissement puis accumulation (fifo).

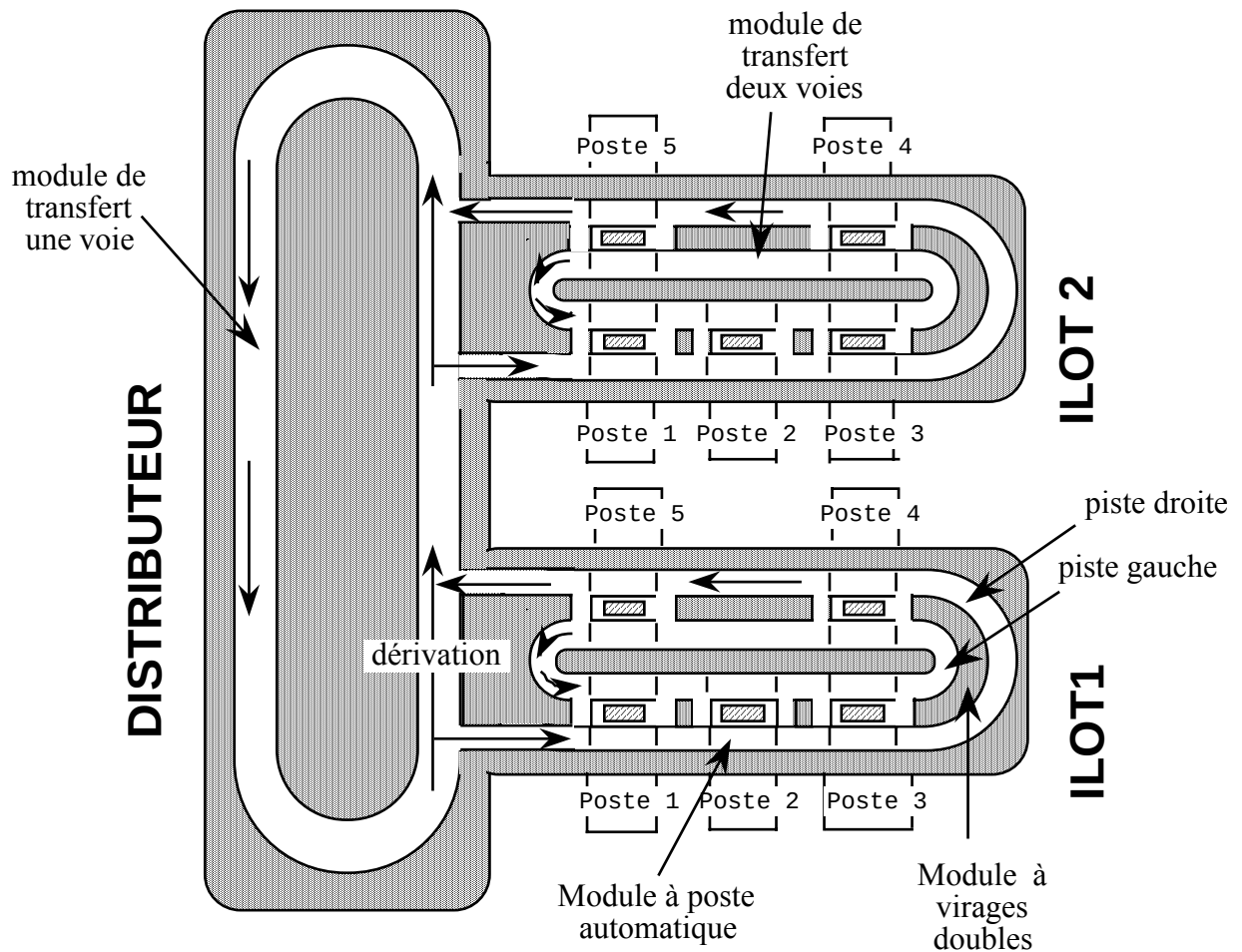


Figure VII.1 - Plan général de l'atelier

1.2. Conduite du système de production

La quantité de palettes qui circule dans l'atelier d'assemblage est fixe. Chaque palette est codée dynamiquement : l'information qu'elle véhicule est modifiée au cours du temps (état du produit, opération suivante,...) par les postes de travail sur lesquels elle passe.

a. Gestion du distributeur

Les règles de gestion de l'anneau de distribution sont les suivantes :

- a) une palette passe de l'anneau de distribution vers un îlot si celle-ci est concernée par l'îlot ;
- b) une palette concernée ne peut entrer dans l'îlot que si le stock amont de réception de cet îlot est apte à la recevoir. Si le test est négatif, la palette reste en attente sur l'anneau de distribution jusqu'à l'occurrence d'une place libre ;
- c) une palette peut sortir d'un îlot et rejoindre le stock central (sur l'anneau de distribution) uniquement si toutes les opérations à subir sur cet îlot ont été réalisées ;
- d) une palette sortant d'un îlot est prioritaire sur celles qui circulent dans le distributeur. On peut toutefois concevoir un rebouclage sur l'anneau intérieur de l'îlot si le stock aval de sortie sur le distributeur est plein.

b. Gestion des îlots

Chaque îlot fonctionne de la même manière. Les règles de gestion sont les suivantes :

- a) les palettes quittant le distributeur entrent dans un îlot par son anneau extérieur. Cette contrainte liée à l'architecture même du système de production impose un passage obligatoire du produit sur le premier poste de l'îlot pour subir ou non l'opération. C'est notamment le cas du produit B (voir paragraphe 4) ;
- b) Un produit est autorisé à quitter l'îlot dès que toutes les opérations qu'il devait y subir sont réalisées ;
- c) A chaque palette est affectée une gamme de fabrication, qui correspond à la suite logique des opérations à subir. Ainsi, pour une gamme donnée, plusieurs routages sont possibles pour le produit en fonction des aléas de production et des règles de gestion.

Au niveau des trajectoires, si le poste suivant logique dans la gamme de fabrication correspond au poste suivant physique (implantation ligne¹³) la palette transitera par l'anneau extérieur. Dans le cas contraire, elle rejoint le poste suivant par l'anneau intérieur.

¹³ Architecture d'un atelier de fabrication selon laquelle la succession spatiale des postes de travail coïncide avec le séquençement des opérations à exécuter sur chaque pièce.

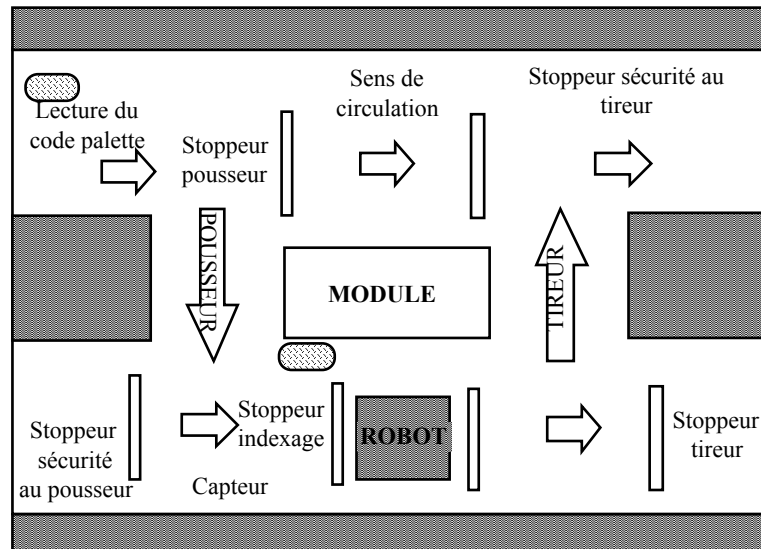


Figure VII.2 - Architecture d'une cellule de production.

L'atelier fonctionne en flux tiré¹⁴. Un stock amont saturé sur un poste entraîne la poursuite du cheminement de la palette sur l'artère centrale. La palette reste sur l'anneau central et cherche le poste le plus "proche" pouvant exécuter la même opération (en fonction des taux d'occupation des machines, afin de les maximiser).

c. Gestion des cellules au sein de l'îlot

La structure physique d'un module de production est montrée en Figure VII.2 et son fonctionnement schématique est illustré en figure VII.3.

- Les blocs *T1* à *T7* matérialisent des portions de convoyage (bandes transporteuses). Elles sont paramétrées par une capacité et un temps de transport ;
- Le bloc Poste regroupe les actions d'indexage, d'opération technologique et de libération du produit à un poste de l'îlot. Cette action sera paramétrée par une durée globale de travail de l'opérateur ;

¹⁴ Un atelier travaille en flux **poussé** si, à l'issue du traitement qu'elle a subi sur un poste, chaque pièce est automatiquement envoyée sur le poste suivant. Le flux est dit **tiré** si, chaque fois qu'une machine est libre, elle "appelle" la prochaine pièce à traiter. Le "juste à temps" est la combinaison de ces deux stratégies. Ces trois stratégies peuvent être modélisées aisément par RdP [Valette...90].

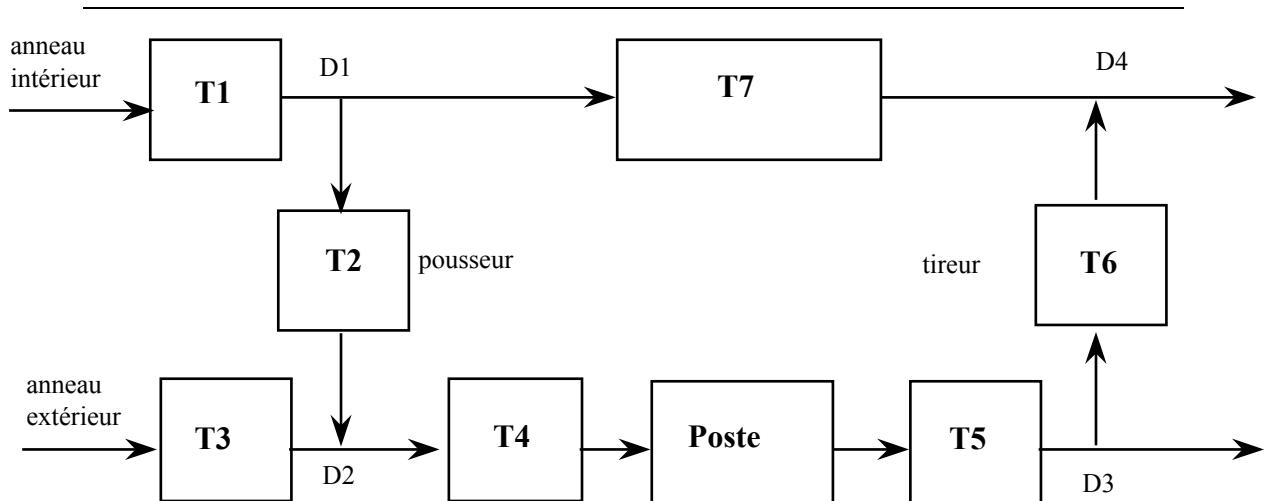


Figure VII.3 - Fonctionnement d'une cellule de production.

- La cellule regroupe 4 centres de décisions *D1* à *D4* dont les règles de fonctionnement sont les suivantes :
 - * **pour D1 et D2** : Un produit venant de *T1* est stoppé et orienté vers *T4* via *T2* s'il est concerné par l'une des opérations du poste. Il est orienté vers *T7* dans le cas contraire ou bien si *T4* est plein. Une priorité est donnée à la palette allant de *T2* vers *T4* par rapport à celle venant de *T3*. On notera que *D2* est prioritaire sur *D1*. Cela signifie qu'en cas d'accumulation des palettes en *T3*, *D1* ne fera rien venir en *T4*, et attendra la libération de *T3* en stoppant les palettes en fin de *T1* ;
 - * **Pour D3** : Un produit sortant de *T5* restera sur l'anneau extérieur si le poste suivant logique (cf gamme) est le suivant physique, mais également si la palette doit rejoindre le distributeur (fin de travail sur l'îlot). Elle bifurquera vers l'anneau intérieur dans le cas contraire ;
 - * **Pour D4** : Les palettes issues de *T6* sont prioritaires vis à vis de celle venant de *T7*.

1.3. Configuration de l'atelier

Chaque élément de l'atelier (tapis roulant, poste de travail,...) est contraint par des paramètres concernant sa capacité (le nombre de palettes qu'il peut contenir à un instant donné) et sa durée de transfert. La valeur de ces paramètres est donnée en Figure VII.4.

élément	capacité	durée
cellule de transport 1 voie du distributeur	4	2 unités de temps par emplacement
dérivation	16	1 unité par emplacement
cellule de transport 2 voies	4 (sur chaque voie)	1 unité par emplacement
cellule de production :		
T1, T3	1	1
T2, T6	1	1
T4	2	1
T5, T7	16	1

Figure VII.4 - Paramètres des composants de l'atelier.

Chaque poste de travail est programmé pour accomplir un ensemble d'opérations, appartenant aux gammes des palettes qui circulent dans l'atelier. Les opérations assurées par chaque poste sont listées en Figure VII.5.

	ILOT 1		ILOT 2	
	Opération 1	Opération 2	Opération 1	Opération 2
Poste 1	Chargement 1er composant de PA sur palette A1		chargement 1er composant de PC sur palette C1	Déchargement de PB B4
Poste 2	chargement 1er composant de PB sur palette B1		Assemblage 2ème composant de PC C2	Assemblage 3ème composant de PB B3
Poste 3	Assemblage 2ème composant de PA A2	Assemblage 2ème composant de PB B2	Assemblage 2ème composant de PC C2	
Poste 4	Assemblage 3ème composant de PA A3	Assemblage 2ème composant de PB B2	Assemblage 3ème composant de PC C3	Assemblage 3ème composant de PB B3
Poste 5	Déchargement de PA A4		Déchargement de PC C4	Déchargement de PB B4

Figure VII.5 - Liste des opérations par poste

L'atelier comprend :

- 50 palettes dans le système. Au lancement, toutes les palettes sont disposées sur l'anneau du distributeur selon un certain ordre. Dès la mise en production, toutes les palettes se meuvent simultanément ;
- 2 opérations technologiques (Op1 et Op2) par opérateur ;
- 1 opérateur (robot) par poste de travail ;
- 1 poste de travail par cellule ;
- 5 cellules par îlot ;
- 2 îlots dans atelier.

1.4. Modélisation de l'atelier

Nous avons choisi de structurer le système de production en restant très proche de la structure physique des éléments de l'atelier. Dans cette décomposition, chaque objet correspond à un élément concret du système de production, par exemple un module de transport ou une cellule robotisée. Toutes les places correspondent à des emplacements concrets (on a cependant opéré quelques regroupements, notamment pour les modules de transport ayant une capacité supérieure à 1). En effet, il n'y a pas lieu dans un tel système de dissocier une fonction de l'organe qui l'assure, et de plus cela favorise la réutilisation des composants.

La présentation de la solution se fera d'une manière "top-down", en présentant d'abord les classes les plus générales, qui sont des classes composites, pour en arriver aux plus primitives. Bien entendu, ce n'est pas dans cette ordre que le processus de modélisation a exhibé les classes du système : la conception Orientée Objet favorise en effet la réutilisation des composants, et la plupart des classes les plus simples (bandes transporteuses, etc...) pourraient être réutilisées dans le contexte d'un atelier complètement différent. Les classes composites de plus haut niveau (*Atelier* et *Îlot*) ont été produites par assemblage ou agrégation d'éléments plus simples déjà définis.

On donne en Figure VII.6 les relations d'héritage et de composition entre les différentes classes du système.

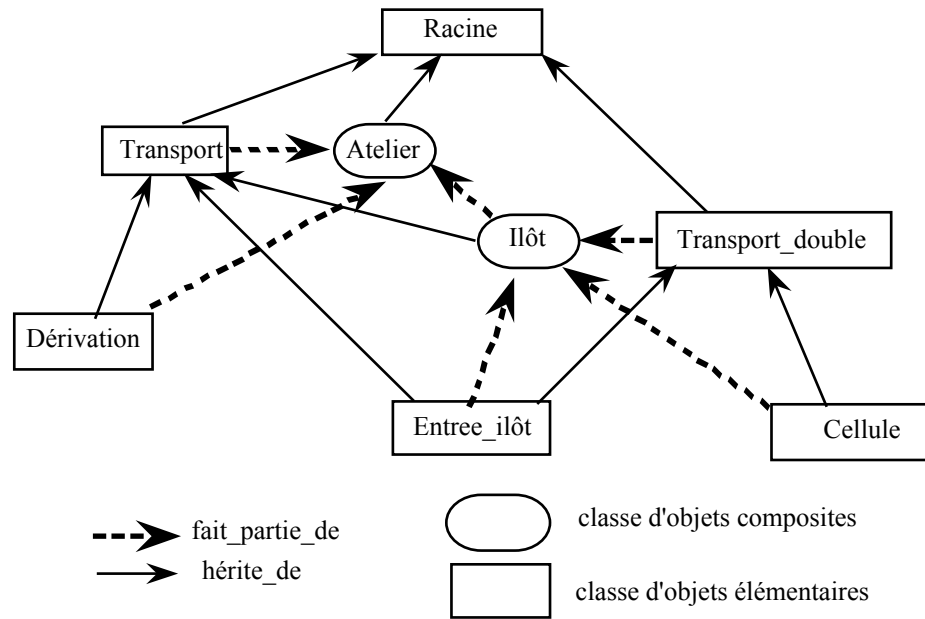


Figure VII.6 - Graphes d'héritage et de composition

a. La classe *Palette*

Une palette portant toujours un et un seul produit, il n'est pas utile d'introduire une classe Produits. La classe *Palette* est une classe d'objets passifs (elle ne publie que des data-flows et a donc un ObCS trivial), et se ramène à une structure de donnée : elle mémorise l'état courant du produit en cours de fabrication, et permet de faire évoluer cet état par l'intermédiaire du service *process*.

Les gammes de fabrication sont ici très simples (une séquence fixe d'opérations, sans alternative). On pourrait très bien envisager de modéliser des gammes plus complexes par un RdP, qui en l'occurrence servirait d'ObCS à la classe *Palette*.

```

Class implementation Palette
Types
    Opérations = (A1, A2, A3, A4, B1, B2, B3, B4, C1, C2, C3, C4)
    Gamme = array[0..3] of Opérations;
Constants
    GammeA : Gamme = [A1, A2, A3, A4];
    GammeB : Gamme = [B1, B2, B3, B4];
    GammeC : Gamme = [C1, C2, C3, C4];
Attributes -- privés
    gamme : Gamme;
    oper : [0..3];
Data-Flows -- publics
    Opération_suivante : Opération is
    do
        result := gamme[oper]
    end

    process is
    do
        ...
        -- la dernière opération d'une gamme correspond au déchargement
        -- du produit et à la réinitialisation de la palette.
        oper = (oper + 1) MOD 4
        -- Lors du déchargement, l'ordonnancement peut également
        -- modifier la gamme de la palette, et donc le
        -- produit à fabriquer
        if (oper = 0) then
            gamme := ...
        end
    end

    create (g : gamme) is
    do
        gamme := g;
        oper := 0;
    end;

```

Figure VII.7 - La classe *Palette*

La classe *Palette* exporte les types utilitaires *Opérations* (un type énuméré) et *Gamme*. Une palette peut indiquer l'opération suivante qu'elle doit subir, par l'intermédiaire du service *Opération_suivante*.

En §VII.3, on a indiqué la configuration initiale de l'atelier, en donnant la quantité de palettes présentes dans le distributeur à l'initialisation du système. Cette configuration initiale sera modélisée par le **marquage initial** des OBCS des objets qui composent le système. Les places de ces OBCS contiendront donc des références d'objet de la classe *Palette*, et la méthode *Create* de la classe *Palette* indique comment initialiser de tels objets.

b. La classe Atelier

Il s'agit de la classe composite correspondant à l'ensemble du système : le système de production que l'on veut modéliser sera défini par un seul objet primitif de la classe *Atelier*.

On a choisi de ne pas identifier le *distributeur* en tant qu'objet composite ; un atelier est donc constitué des objets élémentaires du distributeur, i.e. quatre *Transport* et deux *Dérivations*, et de deux objets composites de classe *Ilôt*.

Dans la mesure où la description de l'atelier ne prévoit pas de montrer l'arrivée des composants et la sortie des produits finis, un *Atelier* est un système clos, sans interface. Sa définition est donnée en figure VII.8

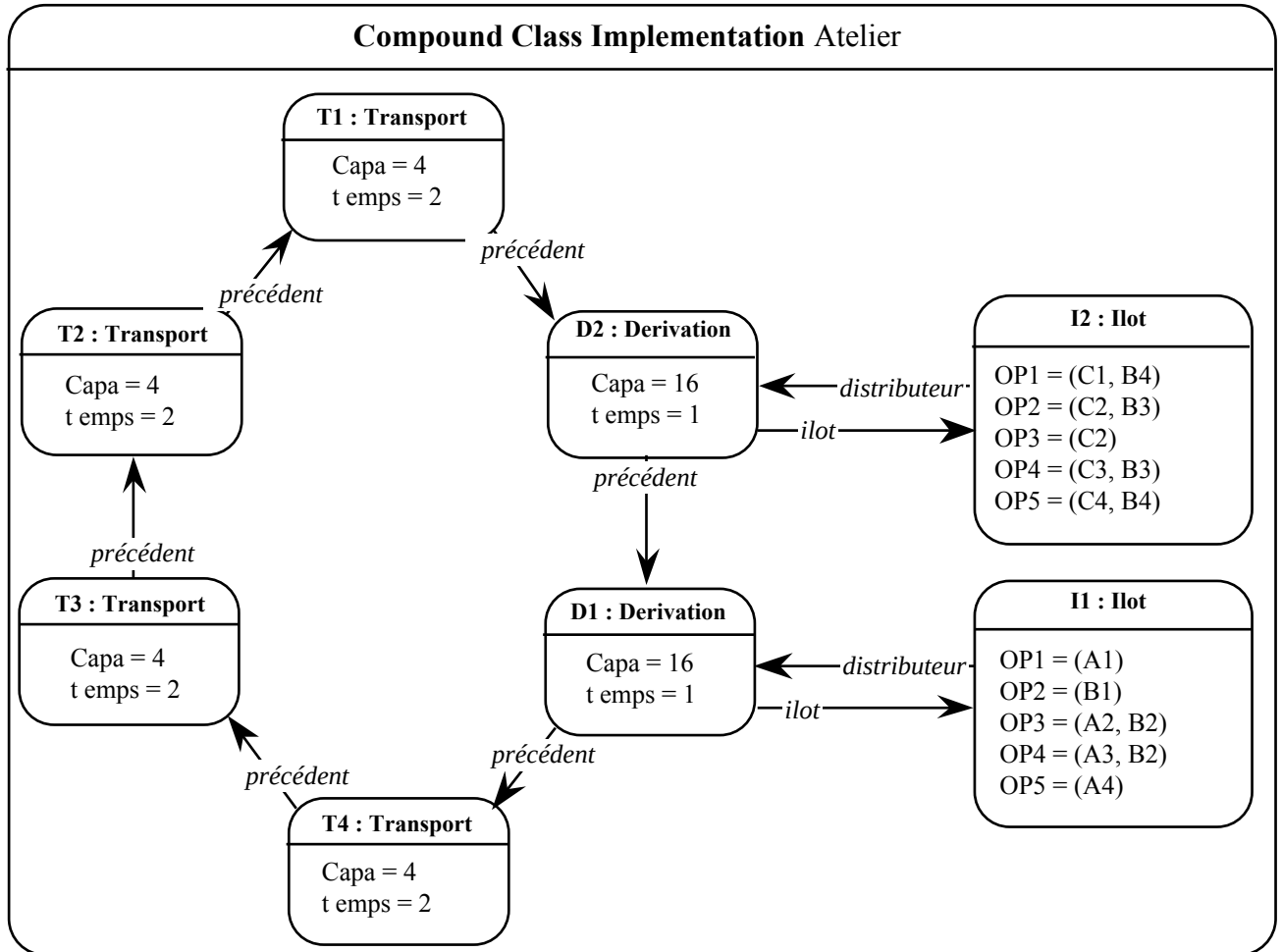


Figure VII.8 - Classe composite *Atelier*

c. La classe *Transport*.

La classe *Transport* modélise le comportement d'une bande transporteuse à une voie, telles que celles qui composent l'anneau de distribution. Sa spécification présente seulement son comportement abstrait, et n'indique donc pas les contraintes temporelles de son utilisation (ici le temps de transit des palettes qui parcourent la section de transport). Cette classe illustre également l'utilisation des places à capacité (cf § II.2.4) et des places triées (cf §II.2.7) : la mention **Hard FIFO** indique que les palettes sont fournies par l'opération *Transmettre* dans l'ordre où elles ont été reçues par l'intermédiaire de la transition *Recevoir*.

Une instance de transport connaît son élément *précédent*, lui aussi instance de transport ou d'une classe plus spécialisée.

A l'initialisation d'une bande transporteuse, on doit indiquer sa capacité **cap** (en nombre de palettes) et le temps élémentaire de transit pour un emplacement **temps** ; le temps minimal de transit d'une palette dans un élément de transport sera donc : $\text{cap} * \text{temps}$.

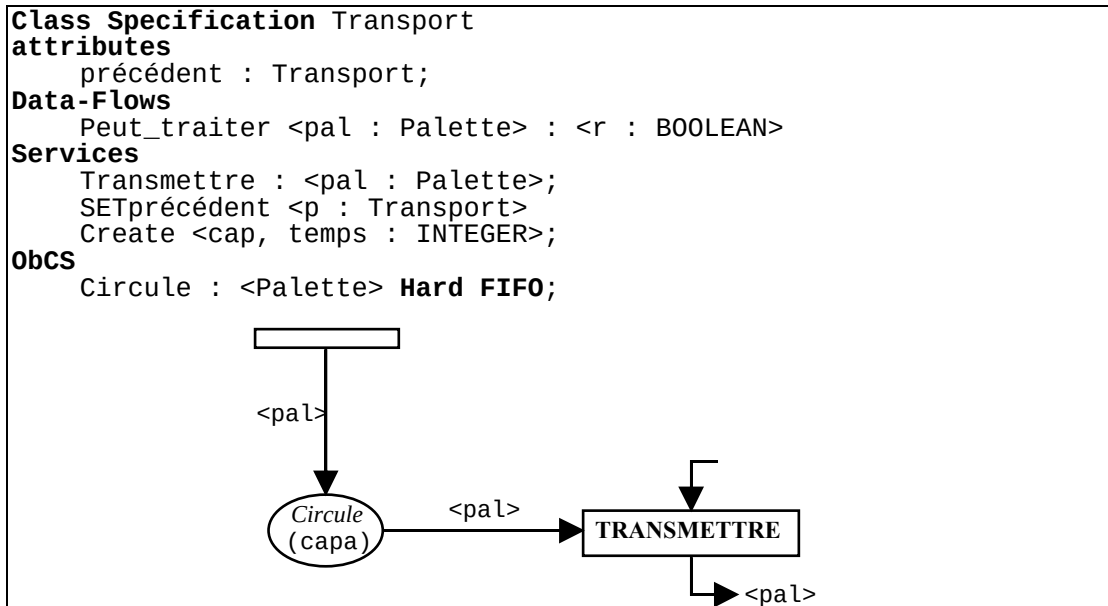


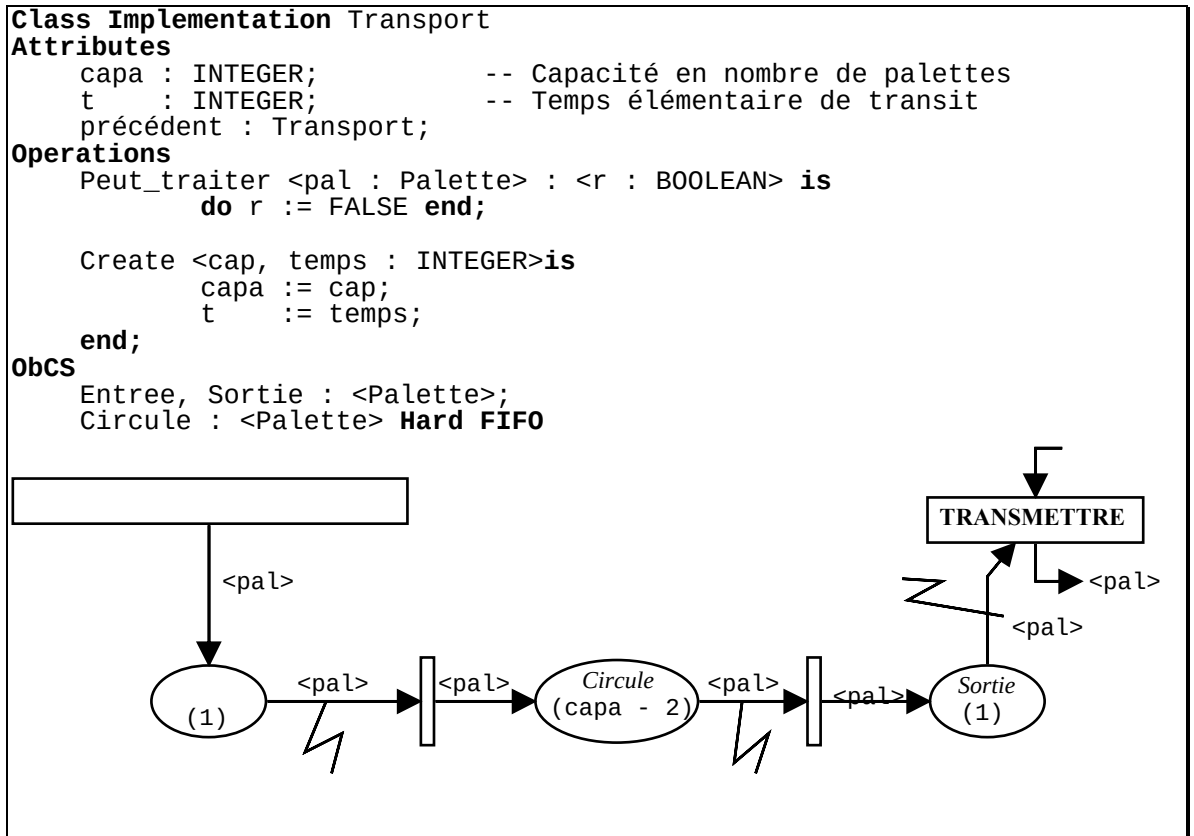
Figure VII.9 - Spécification de la classe *Transport*

La classe *Transport* exporte un service de consultation, *Peut_traiter*, qui définit si la section est en mesure d'effectuer une opération sur la palette passée en paramètre. Cette opération n'est pas utile pour une section de transport simple, mais est introduite afin d'être surchargée par les classes dérivées de *Palette* (notamment la classe *Ilôt*). C'est une notion comparable aux **opérations différées** [Meyer 90b] qui permettent de factoriser des comportements communs en les faisant remonter dans la hiérarchie d'héritage.

L'implémentation de la classe *Transport* montre comment une de ses instances recueille une palette en invoquant le service *Transmettre* à son objet *Précédent*, et s'en décharge en répondant à cette même opération.

L'implémentation indique également les contraintes temporelles d'utilisation des objets de la classe, correspondant ici au temps de transit des palettes. Les places de capacité 1 en entrée et en sortie permettent d'éviter que des palettes n'entrent ou ne sortent de la section à une vitesse supérieure à la vitesse de transit nominale pour un élément.

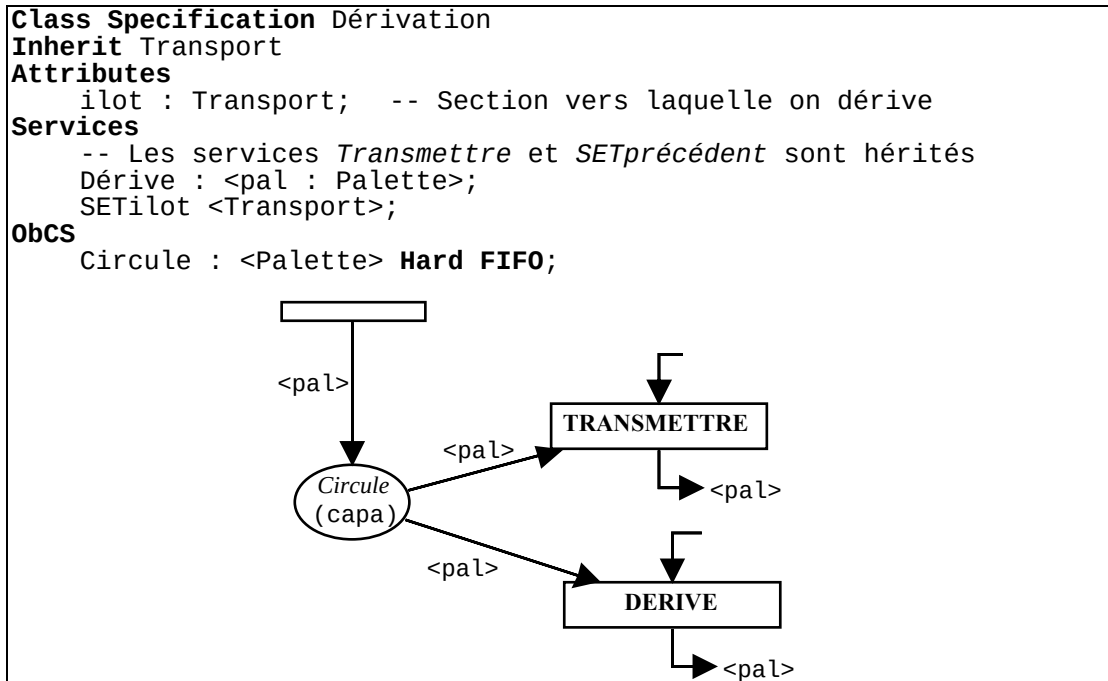
On peut également remarquer que, à l'inverse de la plupart des exemples que nous avons déjà rencontrés, l'implémentation n'a pas compliqué l'opération *Transmettre*, qui reste modélisée comme une simple transition. Ici, (et ce sera le cas dans le reste de cet exemple) l'implémentation a essentiellement pour but de raffiner le comportement interne de l'objet, en rendant plus fine la description de son espace d'états.

Figure VII.10 - Implémentation de la classe *Transport*

d. La classe Derivation

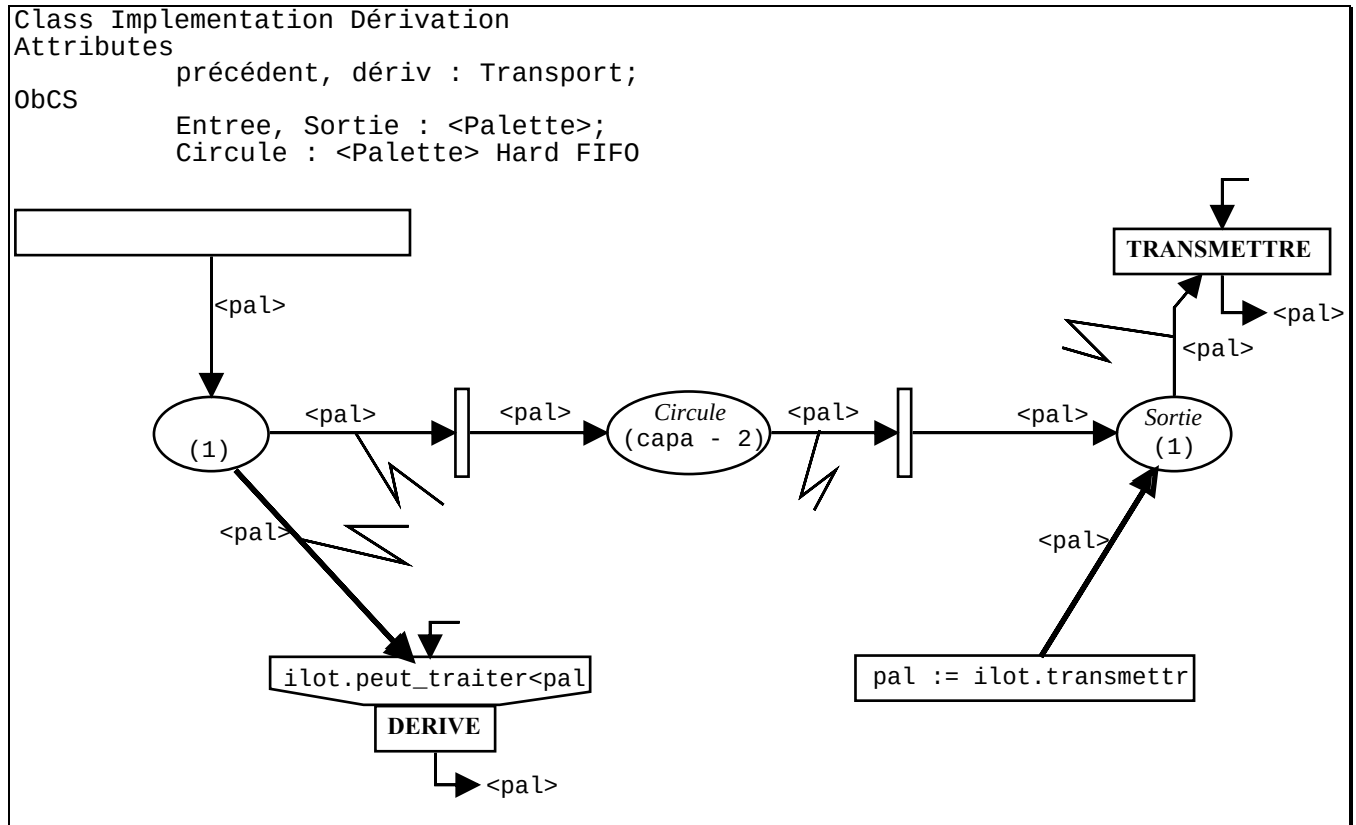
Une dérivation est un élément de transport à une voie, mais qui offre en plus un service permettant de dériver une palette en transit vers un autre élément de transport. On trouve donc ici une utilisation très naturelle de la relation d'héritage entre classes.

Dans l'atelier que nous décrivons, une Dérivation reçoit une palette (venant de *T4* dans le cas de *D1*, et venant de *D1* dans le cas de *D2*), et la transmet soit à l'îlot (*D1* peut transmettre à *I1*, *D2* à *I2*), soit à l'élément de transport suivant (c'est à dire *D2* pour *D1* et *T1* pour *D2*).

Figure VII.11 - Spécification de la classe *Dérivation*

L'implémentation donnée ici d'une section de dérivation est assez spécifique au contexte de l'atelier modélisé : on voit que la dérivation ne peut avoir lieu que si la section de transport vers laquelle on dérive *Peut traiter* effectivement la palette en transit ; ceci correspond à la clause a. des règles de gestion du distributeur.

De même la clause d. est assurée par l'arc prioritaire (cf §II.2.5) en entrée de la place *Sortie*.

Figure VII.12 - Implémentation de la classe *Dérivation*

1.5. Modélisation d'un ilot

Nous avons maintenant décrit les classes d'objets élémentaires qui composent un objet composite de classe *Atelier*. Nous allons décrire un *Ilot* : de quels objets est-il constitué et par quels objets les services qu'il rend sont-ils assurés ?

Tout d'abord, un *Ilot* peut être considéré comme un *Transport* : il sait *Transmettre* des palettes, et dire si il *Peut_traiter* une palette ou non. La spécification d' *Ilot* est donc identique à celle de *Transport*, à ceci près que l'*Ilot* n'assure pas que les palettes soient restituées dans l'ordre FIFO. La clause **Hard FIFO** doit donc être supprimée.

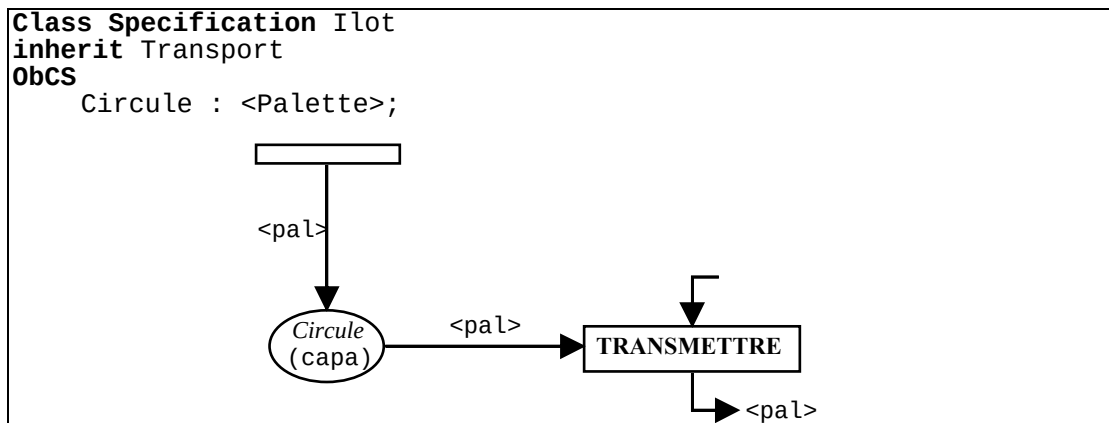


Figure VII.13 - Spécification de la classe composite *Ilot*

Il s'avère que toutes les opérations publiques de *Ilot* sont en fait des opérations de *Entrée_Ilot*. Cela tient au fait que c'est cette classe d'objet qui assure l'interface entre un distributeur et un *Ilot*.

Figure VII.14 - Implémentation de la classe composite *Ilot*

a. La classe *Transport_double*.

Tous les objets qui figurent dans un Ilot ont un objet *Précédent*, et ils ont deux voies de circulation. Nous allons donc définir une classe générique, *Transport_double*, qui jouera vis à vis d'eux le même rôle que *Transport* pour les éléments du distributeur.

L'opération *Peut_traiter* à la même sémantique que pour la classe *Transport*, et sera implantée de façon différente dans chacune des spécialisations de cette classe.

L'opération *Passe_à_droite* indique si la section préfère recevoir la palette sur sa voie de droite, ou de gauche.

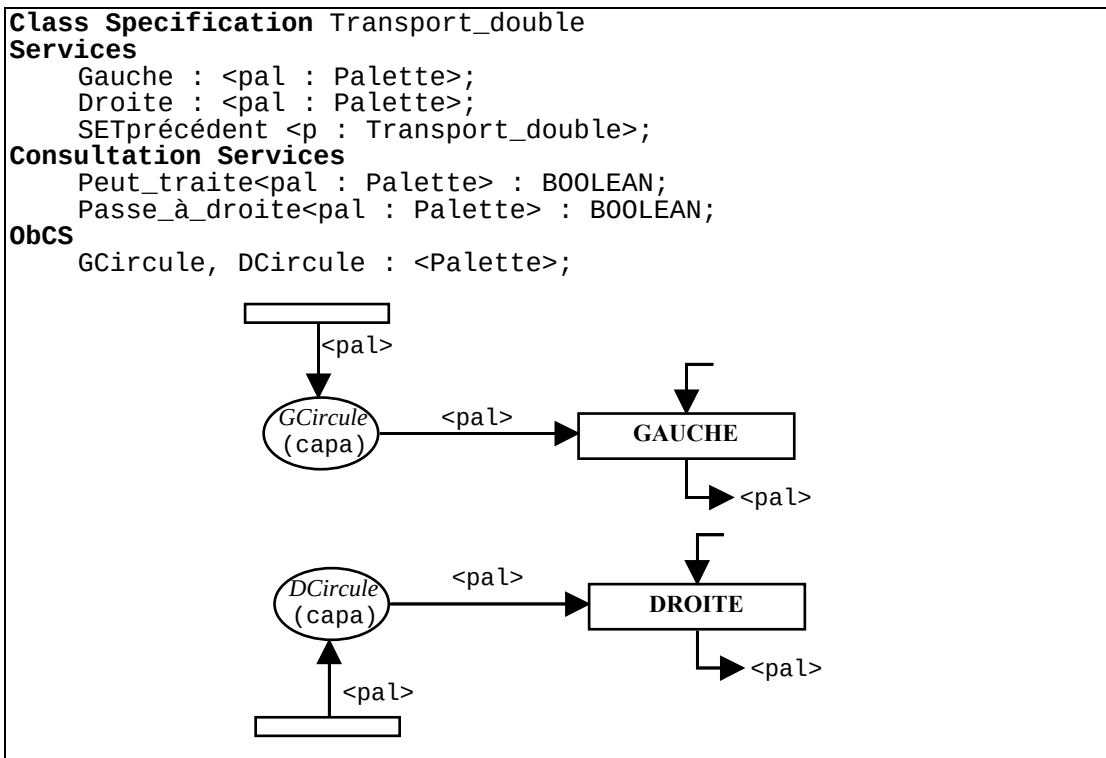


Figure VII.15 - Spécification de la classe *Transport_double*

L'implémentation de *Transport_double* est très simple, reprenant les principes d'implémentation de la classe *Transport*. Elle se contente de faire progresser une palette, et la passe au suivant sur la même voie que celle sur laquelle elle l'a reçue.

Class Implementation Transport_double**Attributes**

précédent2 : Trans_2_voies;

Consultation Services

Peut_traiter<pal : Palette> : <r : Boolean> is

do

r := FALSE;

end -- Peut_traiter

Reste_a_droite<pal : Palette> : <r : Boolean> is

do

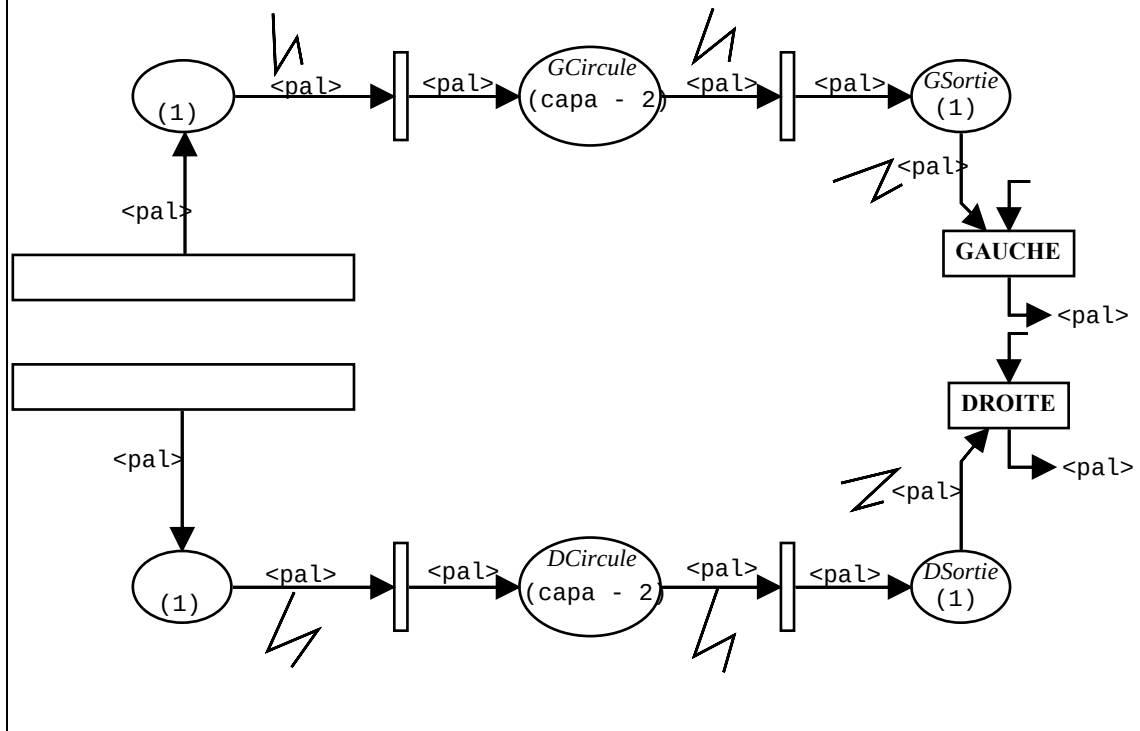
r := Peut_traiter<pal>;

end -- Concerne

obcs

GEntree, GCircule, GSortie : <Palette>;

DEntree, DCircule, DSortie : <Palette>;

Figure VII.16 - Implémentation de la classe *Transport_double***b. La classe Entree_ilot.**

La classe Entree_ilot se comporte à la fois comme un module de transport simple et double, ce qui n'est pas surprenant car elle assure l'interface entre le distributeur (constitué de module de transports à une voie) et les ilots (constitués de modules de transport à deux voies).

Nous sommes donc en présence d'un cas d'héritage multiple (cf §IV.2.d), et il nous faut traiter un conflit de nom, car les classes Transport et Transport_Double publient toutes deux un attribut Précédent. Ceci est fait par un renommage dans la clause d'héritage.

L'ObCS de spécification indique seulement que toute palette reçue deviendra disponible pour un des services *Gauche*, *droite* ou *Transmettre*. On peut vérifier que ce comportement est compatible (au sens défini en §IX.4) avec le comportement des deux classes ancêtres.

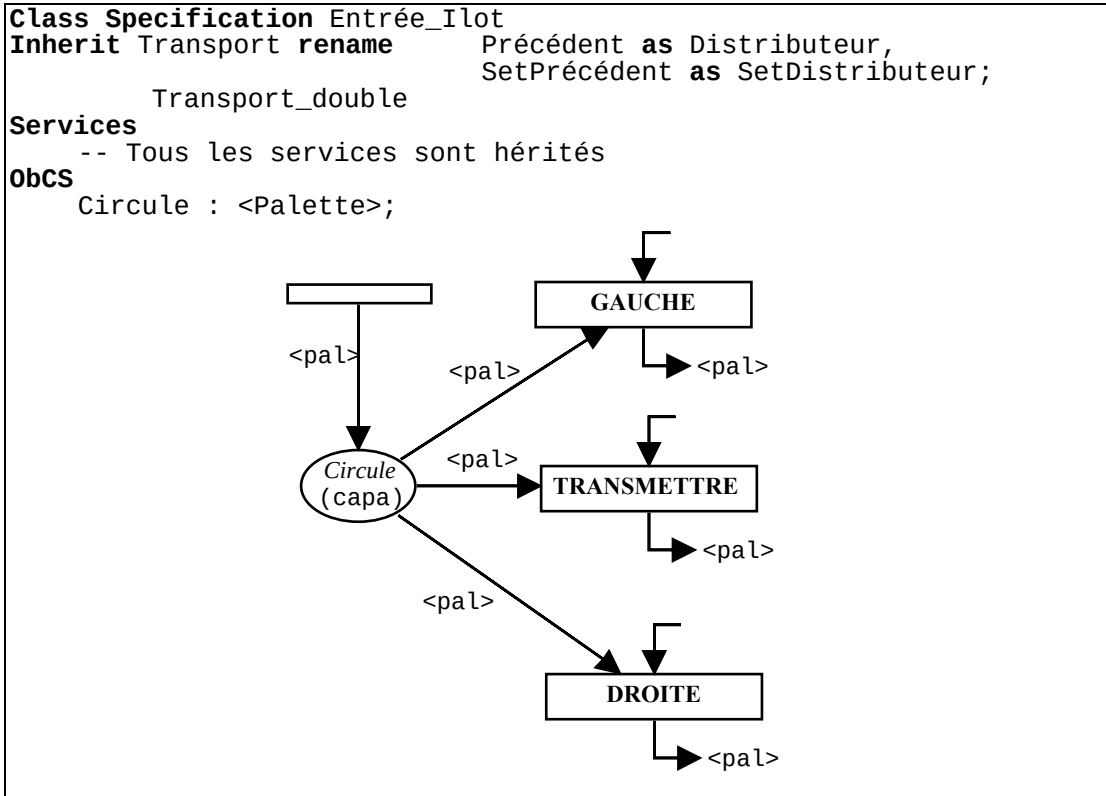
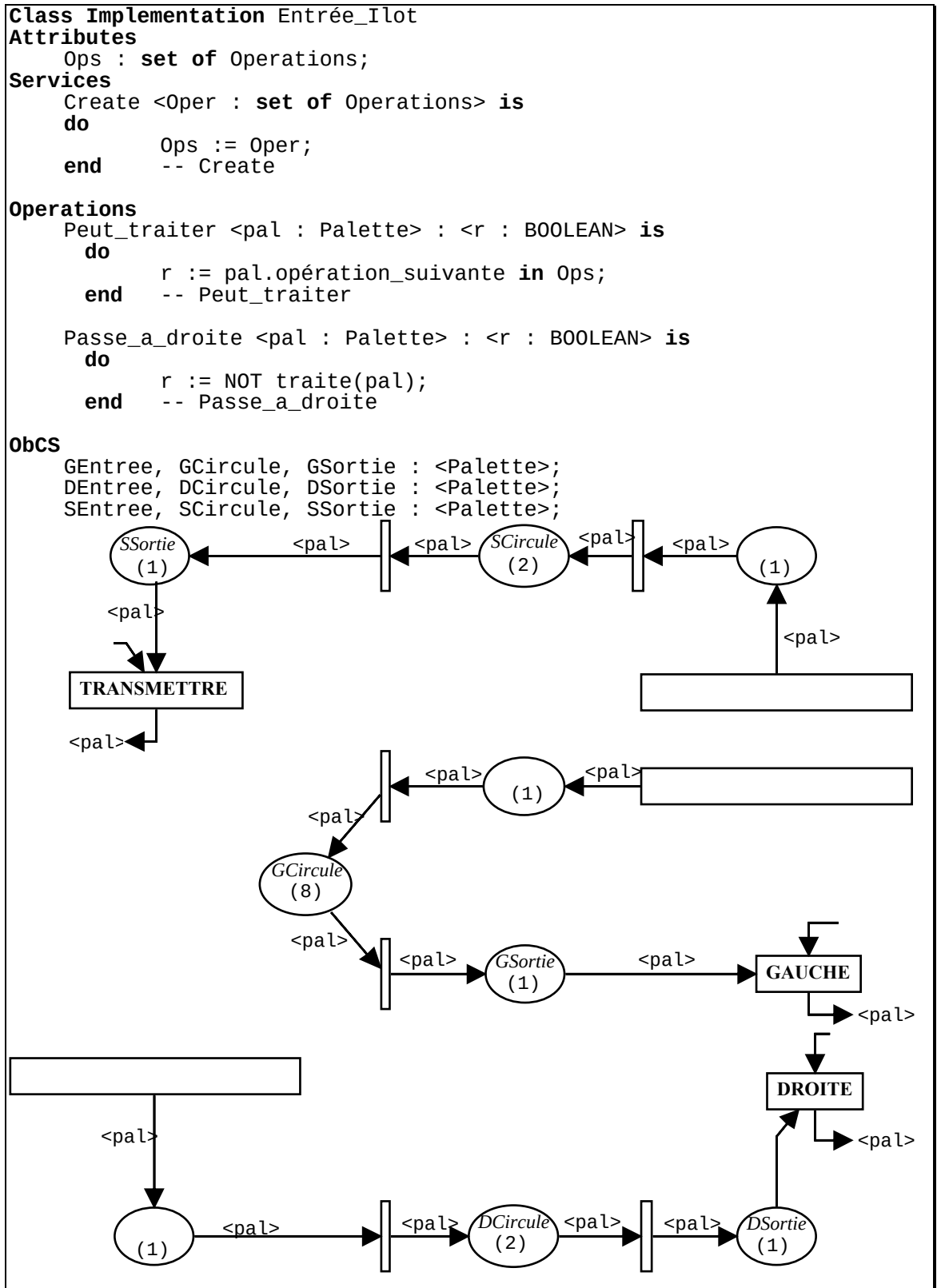


Figure VII.17 - Spécification de la classe *Entrée_ilot*

Figure VII.18 - Implémentation de la classe *Entrée_ilot*

Un objet de cette classe effectue le raccordement entre le distributeur et l'îlot auquel il appartient. Les palettes entrent en provenance du distributeur sur la voie de droite. Après avoir fait le tour de l'îlot, elles retournent dans le distributeur si elles sont sur la voie de droite par appel de *Transmettre*, ou amorcent un autre tour dans l'îlot si elles sont sur la voie de gauche.

On remarque que le service de consultation *Passe_à_droite* est redéfini par rapport à la classe *Transport_double* : une palette doit rester sur la voie de droite si elle a subi toutes les opérations disponibles sur l'îlot ; le test est donc inversé par rapport à une section de transport double normale, où la palette reste à droite si l'élément suivant doit la traiter.

La clause b) de la gestion du distributeur est garantie par le fonctionnement en flux tiré de l'ensemble de l'atelier.

c. La classe Cellule

Cette classe d'objet modélise les cellules contenant les postes de travail robotisés, et hérite de la classe *Transport_double*. Toutefois son ObCS de spécification montre qu'une palette peut changer de voie lors de la traversée d'une cellule (Figure VII.19).

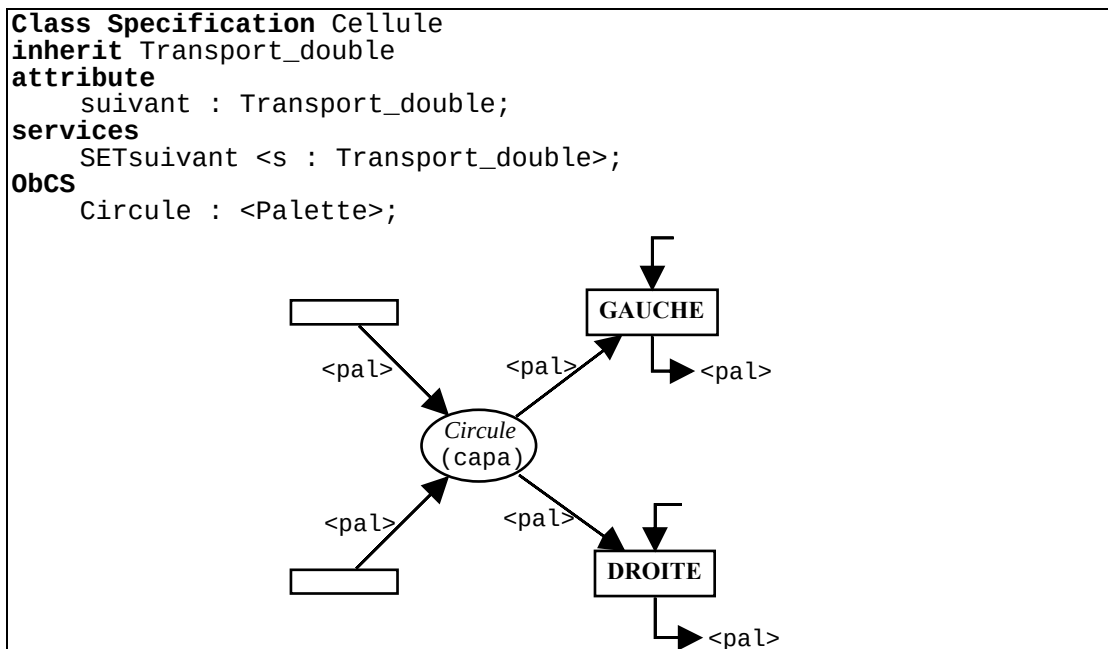


Figure VII.19 - Spécification de la classe *Cellule*

La Figure VII.20 donne l'implémentation de la classe cellule. Le service de consultation *Traite* est redéfini par rapport à la classe *Transport_double* : il teste si l'opération suivante que doit subir la palette peut être assurée par le poste.

```

Class Implementation Cellule
Attributes
    Ops : set of Operations;
Services
    Create <Oper : set of Operations> is
    do
        Ops := Oper;
    end
    -- Create
Data-Flows
    Peut_traiter<pal : Palette> : <r : BOOLEAN> is
    do
        r := pal.opération_suivante in Ops;
    end
    -- Traite
obcs
    Toutes les places sont de type <Palette>

```

Figure VII.20 - Implémentation de la classe *Cellule*

Une palette arrivant par la voie de gauche qui n'est pas concernée par le poste reste sur la voie de gauche; sinon, elle va en *T2* puis *T4* pour subir sur le poste l'opération suivante de sa gamme. Une palette arrivant par la voie de droite passe toujours par le poste. A la sortie du poste en *T5*, elle va vers la voie interne (celle de gauche) si l'on se trouve dans la dernière cellule de production *P5* de l'îlot et qu'il reste des opérations à faire sur l'îlot, ou si le poste suivant ne peut pas effectuer l'opération suivante de la gamme.

On voit donc que la décision de faire transiter la palette vers la voie de gauche dépend d'une condition non locale : la cellule doit consulter l'élément de transport suivant (*suitant.Passe_à_droite*) pour prendre cette décision.

On pourra vérifier que les règles de gestion d'une cellule de production sont bien vérifiées.

2. INTERFACES PERSONNE / LOGICIEL

Nous allons montrer dans cette étude de cas comment le formalisme des Objets Coopératifs s'intègre dans le processus de conception d'une application interactive.

Pour les applications interactives comme pour toute autre application, le processus de conception peut se décomposer dans les étapes classiques d'un cycle de vie que sont l'analyse des besoins, la spécification, la conception et l'implémentation. Des méthodes spécialement adaptées à la prise en compte des besoins de l'utilisateur dans le processus de conception ont été proposées [Barthet 88]. Le formalisme des objets coopératifs, quant à lui, est plus particulièrement adapté à l'étape de conception de l'interface, et pourrait s'intégrer dans les approches méthodologiques citées ci-dessus.

Les progrès très rapide du matériel on donné lieu à une mutation radicale des interfaces proposées aux utilisateurs de logiciel. L'état de l'art est aujourd'hui aux stations de travail graphiques, pourvues d'un dispositif de pointage (la souris) permettant la désignation directe. L'interaction entre l'utilisateur et la machine se fait par interaction avec des représentations symboliques (iconiques) de l'information. Ce type d'interfaces est aujourd'hui suffisamment répandu et bien maîtrisé pour qu'il soit possible de les normaliser, et plusieurs standards sont proposés pour fixer une présentation (look) et un mode d'interaction (feel) commune à plusieurs constructeurs [IBM 89, Motif 89, Open Look 90].

2.1. Le modèle Seeheim

Le modèle Seeheim [Pfaff 85] est un modèle générique de l'interface personne/logiciel, qui a prouvé sa fécondité en offrant un cadre structurant aux diverses recherches dans ce domaine. Il se trouve, de manière plus ou moins explicite, à la base de la plupart des UIMS (User Interface Management Systems ou systèmes de gestion d'interfaces utilisateur) modernes.

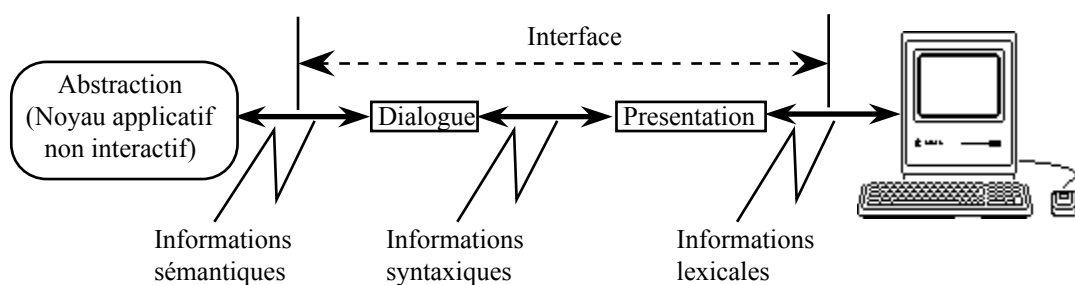


Figure VII.20 - Le modèle Seeheim

Le modèle Seeheim propose une vision linguistique de l'interaction personne/logiciel, en structurant cette communication en niveaux sémantiques, syntaxiques et lexicaux. Cette structuration induit une structure logicielle à trois composants : L'Abstraction, le Dialogue, et la Présentation.

- **L'Abstraction** représente la sémantique de l'application, indépendante de toute interface de manipulation. Dans un cadre Orienté-Objet, l'Abstraction consiste en un ensemble de classes, et

sa conception doit être entreprise par des techniques de Conception Orientée-Objet [Rumbaugh...91, Booch 87] ;

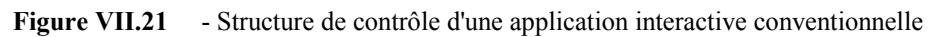
- La **Présentation** est responsable des aspects lexicaux du dialogue, en entrée aussi bien qu'en sortie. Ce composant prend en charge les différentes interactions physiques, et doit transformer les actions de l'utilisateur en unités de dialogue plus abstraites (par exemple transformer l'information lexicale «l'utilisateur a cliqué en coordonnées x,y» en l'information syntaxique «L'utilisateur a pressé sur le bouton Quitter»). Le composant Présentation est aujourd'hui très bien traité par les UIMS disponibles, et sa conception n'est plus à la charge du programmeur d'applications ;
- Le **Dialogue** traite des aspects syntaxiques de l'interaction. Ce composant structure la coopération personne/logiciel à un haut niveau, et assure la distribution du contrôle entre l'utilisateur et le système. Il sert d'intermédiaire entre l'Abstraction et la Présentation. C'est pour la conception de ce composant Dialogue que nous allons utiliser le formalisme des Objets Coopératifs.

Il faut bien veiller à considérer le modèle Seeheim comme un modèle **conceptuel**, et non pas comme un modèle **d'architecture** des interfaces. Les tentatives de structurer une interface suivant ce modèle se sont en effet heurtées à des problèmes caractéristiques des architectures "en couches" : les trois parties se révélaient avoir une architecture isomorphe, créant ainsi des problèmes de couplage et de maintenabilité ; il était le plus souvent impossible de faire évoluer une partie indépendamment des autres. Dans une approche Orientée-Objet, il est plus judicieux de considérer chaque objet comme un "micro-Seeheim" à lui tout seul, et c'est précisément l'approche promue dans le modèle PAC [Coutaz 87].

Il est remarquable de constater que les différents environnements disponibles, bien qu'offrant une interface de programmation très différente (en terme de structures de données manipulées, de fonctions disponibles,...), présentent toutefois une architecture pratiquement identique ; ceci s'explique par leur origine commune, qui remonte aux premiers travaux effectués par l'équipe d'Allan Kay et Adele Goldberg autour du langage Smalltalk [Goldberg...83]. Les caractéristiques principales des UIMS sont les suivantes :

- Orientation Objet : Les différents éléments de l'interface (fenêtre, menus, boutons,...) sont décrits en termes d'Objets (Window Objects ou **widgets**), avec un comportement minimal par défaut. Le programmeur est alors libre de les surcharger par héritage (ou par manipulation de pointeurs de fonctions dans les langages qui n'intègrent pas l'héritage, tels que C).
- Programmation par événements : Les interfaces produites par de tels environnements sont des **systèmes réactifs** (par opposition aux systèmes transformationnels [Pnueli 86]) : ils sont passifs par rapport à leur environnement, et réagissent aux stimuli qu'ils en reçoivent en déclenchant des opérations internes. En règle générale, le contrôle de l'application est **externe**, c'est à dire que l'application ne définit pas ses propres enchaînements de procédures, mais se contente de répondre aux invocations qu'elle reçoit. De même, l'application ne réclame jamais de donnée à l'utilisateur, (ce qui l'entraînerait dans un dialogue bloquant), sauf dans des cas où

Il est très éclairant de comparer l'organigramme d'une application conventionnelle (pilotée par menus, par exemple) avec celui d'une application pilotée par événements (Figure VII.21).



- 212

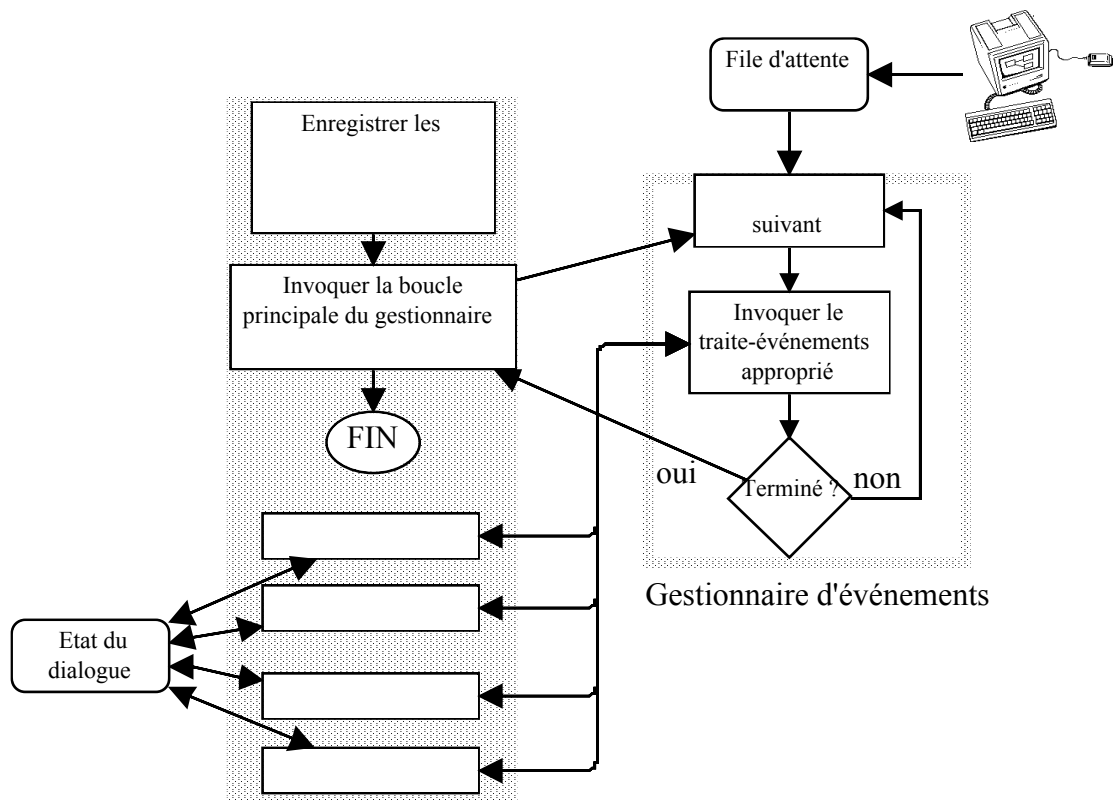


Figure VII.22 - Structure de contrôle d'une application dirigée par événements

- A l'inverse, l'application pilotée par événements présente une structure "à plat", et consiste seulement en un ensemble de procédure de gestion d'événements (event handlers ou **traite-événements**) qui ne s'invoquent pas mutuellement. Un module dédié, le **gestionnaire d'événements** (conceptuellement extérieur à l'application elle-même) gère une file d'attente d'événements, et assure leur interprétation en les ventilant vers le traite-événements qu'ils concernent. Dans la plupart des systèmes modernes la file d'attente d'événements et la boucle qui les traite est totalement invisible au programmeur, qui se contente d'écrire les gestionnaires d'événements, et de les associer explicitement aux différents déclencheurs de l'interface (boutons, menus,...).

On peut qualifier plus précisément la différence entre application conventionnelle et application pilotée par événement :

- L'application conventionnelle présente un **contrôle centralisé** et des **entrées réparties** ; le contrôle est localisé dans la pile des appels de fonctions, et on peut en examinant cette pile déterminer l'historique des appels qui ont conduit à l'état courant. Les entrées sorties sont réparties dans tout le code, car chaque module réclame des entrées à l'utilisateur et produit des résultats.
- L'application dirigée par événements présente un **contrôle réparti** et des **entrées centralisées** ; seule la boucle principale de traitement d'événements effectue des entrées, et les différents

gestionnaires d'événements se synchronisent en accédant en lecture et en écriture à un ensemble de données globales qui définissent l'état courant du dialogue.

On peut résumer cette différence fondamentale en disant que dans une application conventionnelle le contrôle est **fondé sur l'historique** alors que dans une application dirigée par événement le contrôle est **fondé sur l'état** [Cowan...90].

Le problème auquel est confronté le concepteur du dialogue est justement de modéliser cet état, qui représente l'évolution du dialogue entre application et utilisateur ; il lui faut notamment pallier l'absence de structure de contrôle explicite par un formalisme qui lui permette de valider sa modélisation, et notamment de prouver l'absence de blocage (cas où une commande ne peut plus redevenir accessible), ou d'assurer une rétroaction (feedback) informative et permanente : à tout instant, l'interface doit montrer quelles actions sont légales, et lesquelles sont provisoirement inhibées.

Les solutions éprouvées en matière de spécification du dialogue dans les applications conventionnelles (Augmented Transition Networks [Parnas 69, Denert 77] ou grammaires [Olsen 83]) sont malheureusement inadéquates dans le domaine des applications dirigées par événements : elles tendent en effet à modéliser l'utilisateur comme un fichier séquentiel que l'on peut soumettre à une analyse syntaxique, alors que les interfaces modernes souhaitent favoriser un comportement non séquentiel de l'utilisateur, qui peut à volonté interrompre ou différer une tâche, et entretenir des dialogues concurrents avec plusieurs parties de l'application.

2.3. Interprétation du formalisme

Nous nous proposons de donner quelques indications méthodologiques pour guider la conception du dialogue d'une application interactive par les Objets Coopératifs. Dans la suite, nous traitons uniquement de la conception du module Dialogue (en référence au modèle Seeheim) de l'application ; bien entendu, le module Abstraction peut lui aussi être modélisé par Objets Coopératifs, si le domaine d'application s'y prête (par exemple, si l'on modélisait l'interface-utilisateur d'un procédé concurrent).

Une interface sera conçue comme un système d'Objets Coopératifs, dont les objets primitifs publient des **services d'interface** (cf §IV.1.2). Pour compléter sa conception, le modélisateur doit définir des liens entre les widgets, éléments de la Présentation (boutons, menus, etc...) et ces services d'interface, en spécifiant par l'intermédiaire de quels éléments un service d'interface est invoqué. Trois types de relation existent entre widgets et services d'interface :

- relation $1 \rightarrow 1$: un widget est associé à un service d'interface et à un seul. C'est le cas, par exemple, où un service est déclenché par le clic de l'utilisateur sur un bouton, ou par la sélection d'un élément de menu ;
- relation $n \rightarrow 1$: plusieurs widgets sont associés au même service. Un tel cas signifie que plusieurs actions différentes de l'utilisateur provoquent la même évolution du dialogue. L'exemple traité plus loin dans ce chapitre présente une relation de ce type ;
- relation $1 \rightarrow 0$: un widget n'est associé à aucun service d'interface. L'interaction de l'utilisateur avec le widget n'a donc aucune influence sur l'avancement du dialogue. C'est le cas des

éléments de dialogues passifs, tels qu'une zone de texte éditable par exemple ; le fait que l'utilisateur tape du texte dans la zone ne fait pas évoluer le dialogue (le texte qu'il a tapé, quant à lui, pourra être utilisé plus tard dans l'interaction).

Avec cette association entre widgets et services d'interface, on modélise de manière très précise la distribution du contrôle entre l'application et l'utilisateur :

- Dans la plupart des cas, l'avancement du dialogue est piloté par l'utilisateur, qui invoque des services par l'intermédiaire des widgets. Cette invocation provoque le franchissement d'une transition de l'ObCS, qui fait évoluer l'état du dialogue, autorisant éventuellement de nouveaux franchissements et en inhibant d'autres ;
- Le service invoqué peut présenter une OpCS complexe, avec des Transitions Privées (TP) : ces transitions sont déclenchées automatiquement (sous le contrôle de l'application) et modélisent donc des enchaînements obligatoires d'opérations [Barthet 88] ;
- Enfin les dialogues modaux, où l'utilisateur est entraîné dans une entrée/sortie bloquante, sont modélisés comme des opérations internes appelées dans les actions des transitions.

2.4. Exemple

La spécification de l'exemple que nous traitons est très simple : il s'agit de définir une interface générique de manipulation pour des n-uplets stockés dans une table d'une Base de Données (BD) relationnelle. L'interface doit permettre l'ajout de nouveaux n-uplets, la suppression de n-uplets, la sélection et la modification de ceux qui sont déjà stockés dans la table. Le but que l'on se fixe est de fournir un dialogue totalement piloté par l'utilisateur, par opposition aux dialogues pilotés par menus que l'on rencontre dans la plupart des Systèmes de Gestion de Base de Données (SGBD) relationnels. Pour présenter notre solution, nous nous appuyons sur le modèle de Seeheim, en décrivant successivement sa Présentation (ou représentation externe), son Abstraction et son Dialogue.

a. Présentation

L'interface proposée est générique en ce sens que le dialogue qu'elle décrit ne dépend pas de la structure des n-uplets en cours d'édition (nombre et type de leurs champs), et que la même structure de dialogue doit pouvoir être proposée pour toutes les tables de la BD. On suppose simplement que toutes les tables ont un attribut qui leur sert de clé, et dont la valeur sert donc à identifier de manière unique chacun des n-uplets de la table.

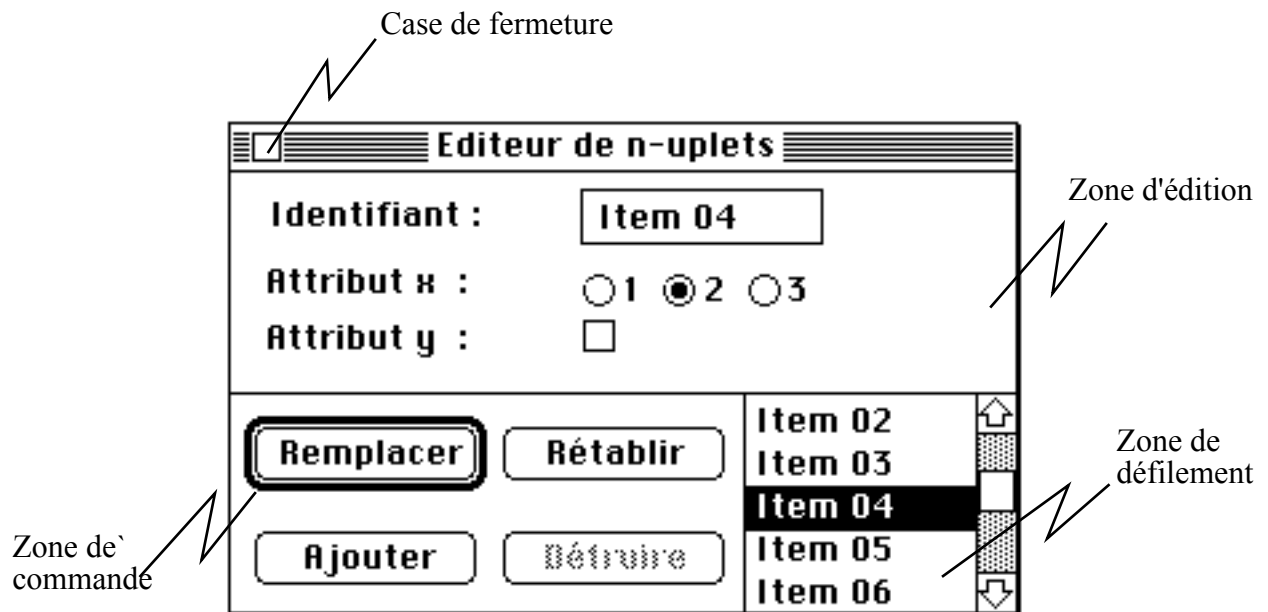


Figure VII.23 - Représentation externe de l'éditeur de n-uplets

La figure VII.23 présente la représentation externe de l'interface. La fenêtre-dialogue présente trois sous-fenêtres distinctes :

- Une **zone d'édition**, dans laquelle les attributs du n-uplet sélectionné peuvent être édités, par l'intermédiaire d'éléments de dialogue standard (boutons radio, cases à cocher, zones de textes...[IBM 89]) ;
- Une **zone de défilement**, qui montre la liste des n-uplets présents dans la table, en affichant uniquement la valeurs de leur attribut-clé. Les éléments de cette liste sont sélectionnables en les pointant avec la souris ;
- Une **zone de commande**, qui permet de déclencher les primitives du SGBD (création , suppression, remplacement) en cliquant sur des boutons poussoir. Nous avons ajouté le bouton Rétablir, qui permet d'annuler les modifications effectuées sur un n-uplet et de restaurer son état initial.

b. Abstraction

Plusieurs choix sont possibles pour identifier l'Abstraction (c'est à dire l'objet "essentiel", indépendant de toute interface de manipulation) qui est sous-jacent à notre application ; on pourrait songer par exemple à décrire un objet "table relationnelle". Pour notre part, nous avons choisi de décrire la partie Abstraction par la classe *n-uplet*, qui décrit les n-uplets stockés dans les tables.

Le fait de nous situer dans le modèle Objet nous permet d'utiliser le concept d'héritage pour obtenir une description générique de l'Abstraction de notre application : La classes *n-uplet* que nous allons décrire va donc être une classe générique, qui n'est pas destinées à être instanciée, mais plutôt à servir d'ancêtre à des classes qui, elles, décriront la structure réelle de n-uplets de la BD.

Cette classe est une **classe d'Objets Passifs** (cf §VI.4) c'est à dire que ses instances se comportent comme de simples structures de données.

```

Passive Class Specification n-uplet
Attributs
    ident : Identifiant -- La clé du n-uplet
Services
    correct :<r : BOOLEAN>; -- Le n-uplet vérifie-t-il les
                           -- contraintes d'intégrité ?

    Affiche <f : Fenêtre>; -- Afficher les valeurs du n-uplet dans f
    Ajoute; -- Ajouter le n-uplet à la table relationnelle.
    Détruit; -- Détruire le n-uplet
  
```

Figure VII.24 - Spécification de la classe *n-uplet*

La figure VII.24 montre la spécification de la classe *n-uplet*. Un n-uplet possède un identifiant (dans l'exemple on considère que l'identifiant peut se représenter comme une chaîne de caractères, mais toute autre représentation serait possible, par exemple une icône ou une image). Le service *correct* permet au n-uplet de vérifier si il répond aux contraintes d'intégrités définies par le modèle de la BD (par exemple assurer la non-duplication des identifiants, ou la vérification de dépendances fonctionnelles). Il faut remarquer que l'évaluation du prédicat *correct* sur une instance de *n-uplet* peut déclencher un message d'erreur **modal**, c'est à dire une fenêtre où l'utilisateur est contraint de signaler immédiatement sa prise en compte de l'erreur avant de poursuivre son dialogue piloté par événements (Figure VII.25)

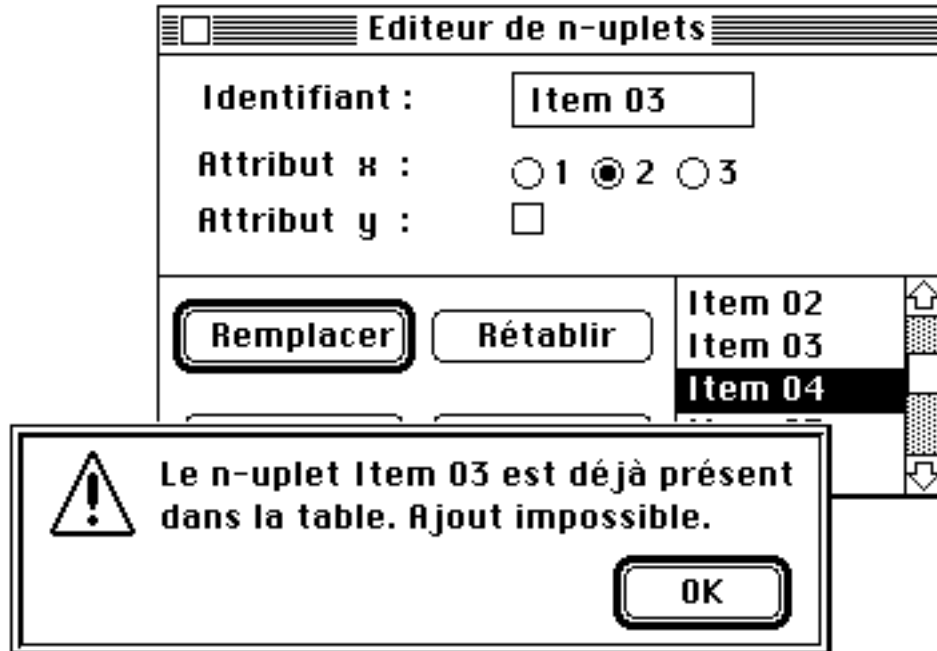


Figure VII.25 - L'éditeur de n-uplets avec un message d'erreur modal

Le n-uplet est en outre capable de s'ajouter ou de se détruire dans sa table relationnelle, et d'afficher ses valeurs dans une fenêtre

c. Dialogue

C'est pour la spécification du Dialogue (ou du Contrôle, suivant la terminologie de J.Coutaz) que l'utilisation du formalisme des Objets Coopératifs se trouve pleinement justifiée. Nous allons modéliser la partie Dialogue du modèle Seeheim par une classe d'Objets Coopératifs, munie de son ObCS, qui va décrire la structure du dialogue et la répartition du contrôle entre utilisateur et application.

Class implémentation Editeur**Attributs**

f : Fenêtre; -- Fenêtre d'édition des attributs du n-uplet.

Services

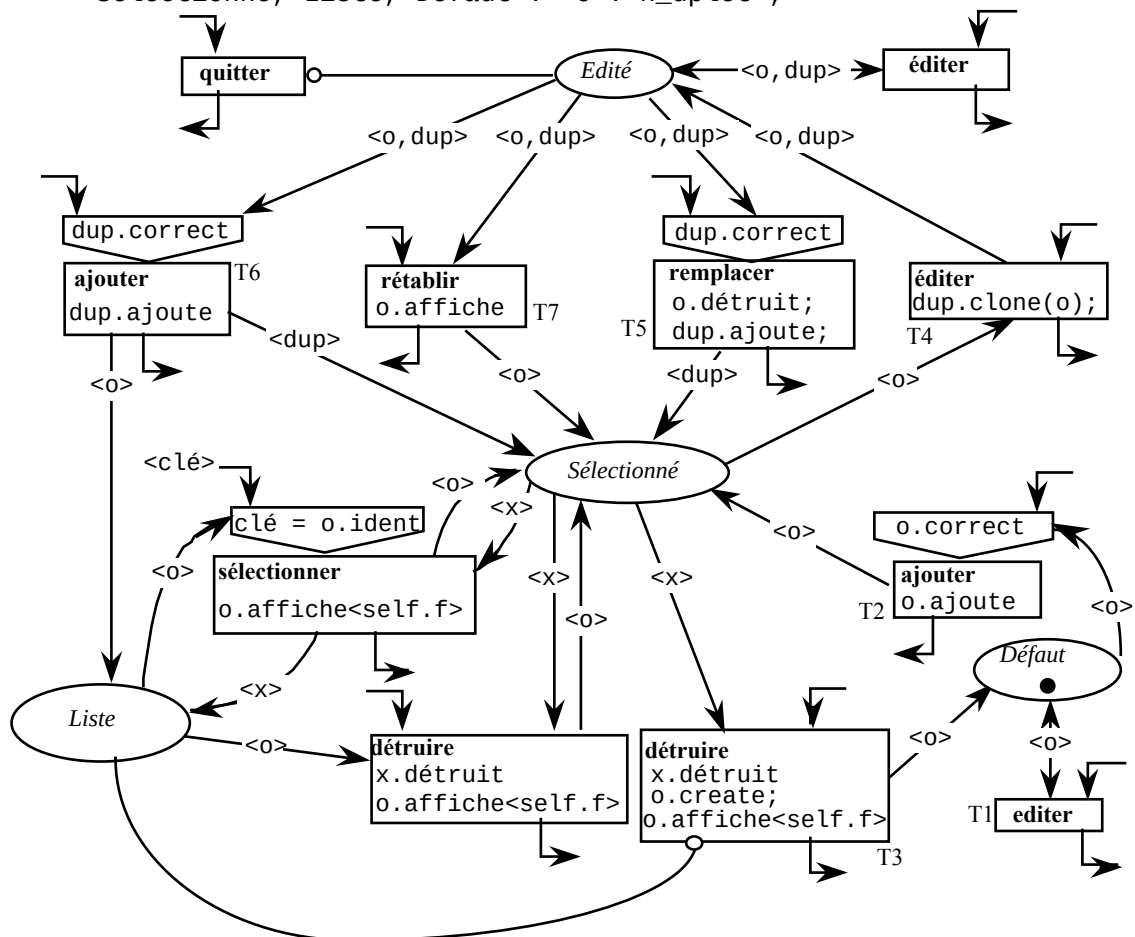
```

Editer; -- Effectuer une opération d'édition quelconque sur
        -- Le n-uplet sélectionné.
Remplacer; -- Remplacer le n-uplet courant par celui en cours
            -- d'édition.
Ajouter; -- Ajouter le n-uplet en cours d'édition
          -- et le sélectionner.
Détruire; -- Détruire le n-uplet en cours d'édition.
          -- En sélectionner un nouveau s'il en reste, sinon
          -- en créer un avec des valeurs par défaut.
Rétablir; -- Annuler toute édition effectuée sur le n-uplet, et
          -- restaurer ses valeurs initiales.
Sélectionner<clé : Identifiant>;
          -- Sélectionner le n-uplet ayant la clé spécifiée,
          -- et afficher la valeur de ses attributs.
Quitter; -- Terminer le dialogue, fermer la fenêtre.
Create(fen : Fenêtre) is
    f := fen; Défaut := n_uplet.Create;
end; -- Create

```

ObCS

Edité : <o, dup : n_uplet>;
 Sélectionné, Liste, Défaut : <o : n_uplet>;

Figure VII.26 - Implémentation de la classe *Editeur*

Pour compléter la définition du Dialogue, il faut associer les divers déclencheurs présents dans la représentation externe aux services offerts par la classe *Editeur*.

- Les quatre boutons-poussoir de la zone de commande (*Ajouter*, *Détruire*, *Remplacer*, *Rétablir*) sont associés aux services de même nom ;
- La zone de défilement est associée au service *Sélectionner*. L'élément de dialogue fournit en paramètre du service l'identifiant du n-uplet désigné ;
- La case de fermeture (en haut et à gauche du cadre de la fenêtre) est associée au service *Quitter*.
- Enfin, tous les éléments de dialogue de la zone d'édition sont associés au service *Edité*. Ainsi, une action quelconque sur un de ces éléments (qui a pour effet de modifier la valeur d'un des attributs du n-uplet) indique au Dialogue que le n-uplet a été édité.

Une fois ces associations établies, on peut donner un aperçu de la dynamique du dialogue en décrivant comment les actions de l'utilisateur font évoluer le marquage de l'ObCS, et réciproquement comment ce marquage active ou inhibe les déclencheurs de l'interface :

- Avec le marquage initial décrit par l'opération *Create*, on dispose d'un n-uplet avec des valeurs par défaut dans la place *Défaut*. Seuls sont actifs les éléments de la zone d'édition (la transition T1, associée au service *Editer*, est franchissable), et le bouton *Ajouter* (transition T2) ;
- Après avoir fixé les valeurs désirées pour le n-uplet, l'utilisateur invoque le service *Ajouter* (transition T2) qui fait passer le n-uplet dans l'état *Sélectionné* ;
- A partir de cet état, l'utilisateur peut invoquer le service *Détruire* (transition T3), qui le ramène au marquage initial. Plus probablement, il choisira d'*Editer* (transition T4) le nouveau n-uplet, stockant ainsi dans la place *Edité* le couple <0, dup> qui désigne respectivement le n-uplet initial et celui résultant de la modification ;
- Après quelques actions d'édition successives, il pourra choisir de *Remplacer* (transition T5) le n-uplet courant par celui qu'il a édité, d'*Ajouter* (transition T6) un nouveau n-uplet avec les nouvelles valeurs d'attribut ou de *Rétablir* (transition T7) les valeurs initiales du n-uplet.

Après quelques itérations de ce processus de modification et d'ajout de n-uplets, on se retrouve dans l'état illustré par la figure VII.23 : la place *Liste* contient un certain nombre de jetons de type <n_uplet>, les places *Sélectionné* et *Défaut* sont vides, et la place *Edité* contient un jeton, indiquant ainsi qu'un n-uplet est en cours de modification. Dans la zone de commande, seuls les boutons *Remplacer*, *Ajouter* et *Rétablir* sont activables, car les Transitions de Service auxquelles ils sont reliés (T5, T6 et T7 respectivement) sont franchissables.

La figure VII.25 montre un message d'erreur déclenché par l'évaluation de la précondition de la transition T6, dans le cas où le n-uplet ne vérifie pas les contraintes d'intégrité.

La spécification du dialogue que nous fournissons présente par ailleurs certains choix arbitraires, qui peuvent être discutables sur le plan ergonomique : Le service *Quitter*, par exemple, n'est activable que si aucun n-uplet n'est édité (i.e. si le marquage de la place *Edité* est vide, ce qui est spécifié par l'arc

inhibiteur). Il pourrait être plus judicieux dans ce cas de demander une confirmation à l'utilisateur, ce qui serait représenté dans l'ObCS par une précondition remplaçant l'arc inhibiteur.

L'un des intérêts de notre approche est justement de permettre de telles discussions en autorisant un prototypage de la dynamique de l'application, tout en offrant une spécification à la fois formelle, concise et complète de la structure du dialogue.

Un autre intérêt est de profiter des possibilités d'analyse des RdP pour vérifier des propriétés de notre modélisation : Il est ainsi trivial de démontrer que les places *Sélectionné* et *Edité* sont bornées à 1, et qu'elles sont mutuellement exclusives, établissant ainsi la propriété souhaitable que l'on n'édite qu'un seul n-uplet à la fois. On peut également démontrer que quel que soit le marquage atteint, une et une seule parmi les transitions associées au service *Editer* est franchissable, montrant ainsi que l'utilisateur peut interagir avec les éléments de la zone d'édition quand il le souhaite.

Chapitre VIII. Exécution d'un modèle d'Objets Coopératifs

Une fois construit le WSCS, le modèle d'un système peut être exécuté par un interprète de réseau prédicat/transition, permettant ainsi au concepteur d'expérimenter la dynamique de son système.

De nombreux interprètes de RdP de haut niveau sont disponibles, certains distribués par des sociétés privées [Techlog 89], et d'autres étant du domaine public¹⁵.

On pourrait envisager, à partir de la description des classes et de la définition du système, de générer directement le WSCS sous la forme des structures de données acceptées par ces outils pour la description des réseaux qu'ils traitent. Cette solution aurait pour inconvénient que les résultats obtenus par l'exécution d'un modèle seraient donnés en termes du WSCS, et qu'il pourrait alors être difficile de les interpréter en termes d'objets et de classes.

Les travaux de [Taubner 87], [Buetler...89] visent à améliorer l'efficacité des interprètes de RdP par une implémentation distribuée, utilisant un algorithme parallèle. Une autre possibilité est de coordonner l'activité de plusieurs interprètes, de telle sorte que chacun exécute une partie du réseau global. Cette solution est judicieuse quand les sous-réseaux modélisent chacun une partie bien distincte du système : ces réseaux sont alors exécutés par leur propre processeur, tout comme le sous-système qu'ils modélisent. Si le système que l'on modélise est réparti, les interprètes peuvent l'être également, de façon à simuler également les communications dans le système. Cette décomposition en sous réseaux significatifs est le résultat naturel d'une modélisation conduite en s'appuyant sur l'approche Orientée-Objet.

En plus de la solution qui consiste à interpréter le WSCS, deux autres stratégies d'exécution des modèles peuvent être proposées, avec un interprète par objet, ou un interprète par classe d'objets.

Quelle que soit la solution choisie, il est nécessaire de disposer de primitives de communication inter-processus (IPC) afin que les interprètes puissent communiquer. Les interprètes communiquent par échanges de jetons, et on doit donc définir des primitives permettant cet échange. Nous allons voir que deux primitives suffisent :

- La première permet à un interprète de déposer un jeton dans une place d'un réseau pris en charge par un autre interprète ;
- La seconde permet à un interprète d'accéder aux valeurs d'un jeton dans une place d'un réseau pris en charge par un autre interprète, en lecture seulement.

¹⁵ Une liste d'outils supportant l'utilisation des RdP est décrite dans [Feldbrugge...87], et maintenue à jour en permanence.

1. UN INTERPRETE PAR OBJET.

Avec cette stratégie d'exécution, chaque objet du système est pris en charge par son propre interprète, qui exécutera son ObCS. Avec cette solution, la dynamique et la topologie du modèle en cours d'exécution présente la même structure que le système modélisé, mais cette approche présente en revanche les inconvénients liés à la prolifération des interprètes quand des objets sont créés dynamiquement dans le système, et d'un trafic plus intense d'IPC.

Les réseaux à exécuter dans cette stratégie sont ceux obtenus après l'étape 3 de la construction du WSCS.

Pour exécuter un modèle suivant cette stratégie, les règles suivantes doivent être appliquées :

- 1° Pour chaque objet primitif, créer un processus interprétant l'ObCS de l'objet, avec le marquage initial défini par le système. Ce processus est identifié par le nom de l'objet ;
- 2° Chaque occurrence de la transition d'appel (générée par l'expansion d'une transition d'invocation, cf §III.1.f) exécute la première primitive de communication pour déposer un jeton dans la place-paramètre correspondante du réseau référencé par le nom de l'objet serveur ;
- 3° De même, chaque occurrence d'une transition de retour dépose le jeton résultat dans la place-résultat correspondante du réseau référencé par le nom de l'objet client (il faut se souvenir que l'identificateur d'appel, envoyé par le client avec les paramètres du service, est de la forme :
Nom_de_classe#Numéro_d'instance#Numéro_d'ordre,
et que l'identité du client est donc *Nom_de_classe#Numéro_d'instance*, cf.§V.1);
- 4° Chaque occurrence d'une Transition d'Accès (TA) doit accéder aux valeurs des attributs stockés dans les places-attributs du réseau référencé par l'objet serveur, en utilisant le deuxième type de primitive de communication.
- 5° Chaque occurrence d'une transition qui invoque l'opération *Create* génère un nouveau processus-interprète qui exécute l'ObCS de l'objet nouvellement créé ; Ce processus est identifié par le nom du nouvel objet.
- 6° La règle de tir d'un interprète est de ne pas autoriser les franchissements concurrents dans son réseau.

2. UN INTERPRETE PAR CLASSE D'OBJETS.

Cette stratégie n'utilise qu'un seul processus pour chaque classe du système. L'avantage principal est ni le nombre de processus, ni la structure des relations que ces processus entretiennent n'évoluent.

Les réseaux que l'on exécute dans cette stratégie sont les ObCS d'extension (§V.3), en suivant les règles suivantes :

- 1° Créer un processus pour chaque classe du système, qui interprète son ObCS d'extension, avec le marquage initial défini en §V.3.4. Un interprète est référencé par le nom de la classe dont il exécute l'ObCS d'extension ;
- 2° Chaque occurrence d'une transition d'appel dépose un jeton dans la place-paramètre correspondante du processus référencé par le nom de la classe de l'objet serveur. De même, chaque occurrence d'une transition de retour dépose le tuple résultat dans la place résultat du processus référencé par le nom de la classe de l'objet client ;
- 3° Une occurrence d'une transition invoquant l'opération *Create* dépose le n-uplet de paramètres dans la place-paramètres de la transition *Create*, dans l'ObCS d'extension de la classe que l'on veut instancier ;
- 4° La règle de tir concurrent est celle du WSCS (§V.4) : on ne peut franchir concurremment que des instances de transitions pour lesquelles la variable *self* est liée à des noms d'objets différents.

Suivant cette stratégie, les communications initiées par les transitions d'appel et de retour sont toujours dynamiques. Ce n'est plus le cas si on utilise un interprète pour l'ObCS de chaque hiérarchie de classe (Etape 6 de la construction du WSCS).

La technique décrite par [Bako 90] pour l'exécution par compilation partielle des RPO est applicable aux ObCS des Objets Coopératifs, à condition toutefois de ne pas utiliser l'accès aux attributs publics d'une classe : cette technique de compilation exclut en effet le recours à la deuxième primitive d'IPC.

Dans une implémentation répartie d'un modèle d'Objets Coopératifs qui utilise les primitives de temporisation décrites en §VI.1.2, des problèmes de **dérive d'horloge** peuvent se produire. Dans le cas d'un appel gardé, si l'horloge de l'appelé est en retard par rapport à celle de l'appelant, le délai de garde sera d'autant moins long, et réciproquement si l'horloge de l'appelant est en retard. Si l'évolution du registre temporel n'est pas décrite dans le réseau (par les techniques de [Richter 85]) mais au contraire gérée par l'environnement, on peut utiliser les techniques de synchronisation d'horloges réparties décrites pour les applications temps-réel [Kopetz...87].

Chapitre IX. Analyse d'un modèle d'objets coopératifs

Si on souhaite voir fonctionner les objets d'un système comme l'entend le concepteur, il est nécessaire que leurs ObCS satisfassent certaines conditions. L'usage des RPO pour définir les ObCS nous permet d'utiliser les techniques d'analyse développées pour ce formalisme afin de prouver un certain nombre de propriétés.

Les propriétés d'un ObCS concernent les aspects suivants :

- Quel type de comportement doit-on attendre d'un serveur «fiable» ? Il doit produire un résultat pour toute invocation qu'il a acceptée, ne jamais générer de résultats non réclamés, et respecter son mode d'emploi (c'est à dire sa spécification) ;
- Les relations entre spécification et implémentation sont claires en ce qui concerne sa structure de données et les services qu'il offre (cf §IV.2.1). En ce qui concerne sa structure de contrôle, on doit assurer la cohérence des ses ObCS de spécification et d'implémentation ;
- De même, la relation d'héritage entre classes est précisément définie en ce qui concerne les structures de données et les services. En ce qui concerne la structure de contrôle, l'ObCS d'une classe spécialisée doit être cohérent avec celui de la classe dont il descend ;

Pour établir ces propriétés, nous allons donner des contraintes portant sur le comportement des ObCS, qui peuvent être vérifiées à posteriori, une fois l'ObCS construit.

Les contraintes que nous donnons ne sont pas définies au niveau du WSCS, mais au niveau des ObCS des objets eux-mêmes, en accord avec la sémantique intuitive de leur ObCS décrite en §III, et ne sont pas liées à la structure des communications entre objets.

Ces contraintes peuvent de la sorte être vérifiées dès la conception de la classe, sans référence aux systèmes dans lesquels la classe est susceptible d'être utilisée.

La formulation de ces contraintes fait référence aux RPO des classes, c'est à dire à leur ObCS amputé de la fonction de disponibilité Disp (§III.3.6) qui définit l'association entre services offerts et transitions de service (TS) du RPO. De même, les Transitions d'Invocation (TI) sont considérées comme des transitions ordinaires, et les objets serveurs sont supposés ne jamais soulever de problème.

Cette formulation a donc pour conséquence que la valeur des paramètres fournis par les clients aux TS ne peuvent pas être pris en compte, ni la valeur des paramètres de sortie des services invoqués. De plus, toutes les instances d'une classe ont la même distribution initiale de jetons, mais avec des valeurs initiales différentes. Ainsi, pour que les contraintes soient significatives pour toute instance, elles doivent être définies sans référence aux valeurs des jetons, et les préconditions et actions des transitions sont donc ignorées (les réseaux sont examinés au second niveau du §II.1.3).

1. DEFINITIONS PREALABLES

Définition IX.1 - Séquences franchissables de transitions

Soit $N = (C, T, V, P, \text{Pré}, \text{Post})$ le RPO de l'ObCS d'une classe, et M sa distribution initiale de jetons.

T^* est l'ensemble des séquences finies de transitions de T .

$L(N, M) = \{ \sigma \in T^* ; M \xrightarrow{\sigma} \}$ est l'ensemble des **séquences franchissables de transitions**

Soit $\sigma \in T^*$ et $T \supset T'$;

$\sigma | T' \in T'^*$ est la sous-chaîne de σ obtenue en enlevant de σ les occurrences de $T - T'$, et

$L(N, M) | T' = \{ \sigma | T' ; \sigma \in L(N, M) \}$.

Si $\sigma \in T^*$ et $t \in T$, alors $\underline{\sigma}(t)$ est le nombre d'occurrences de t dans σ .

Pour un service s accepté par un serveur, on notera **sac** sa transition d'accord, et **sret** sa transition de retour. **Tac** et **Tret** sont respectivement les ensembles des transitions d'accord et de retour de l'ObCS d'un serveur.

2. VERIFICATION DE L'OBCE D'UN SERVEUR

Les contraintes suivantes visent à assurer qu'un serveur se comporte toujours correctement du point de vue de ses clients.

Dans l'ObCS de spécification, un service offert apparaît comme une seule transition, et le fait d'accepter une requête ne peut pas être dissocié de celui de renvoyer un résultat. Ce n'est plus le cas dans l'ObCS d'implémentation, où l'exécution d'un service s'étend sur plusieurs transitions.

Afin que l'on soit assuré qu'un serveur renvoie un résultat si et seulement s'il accepte une requête, son ObCS d'implémentation doit satisfaire les deux critères suivants :

Définition IX.2 - Critère d'honnêteté

Un serveur sera dit **honnête** s'il peut toujours fournir un résultat pour une requête acceptée, sans avoir pour cela à accepter d'autres requêtes :

$$\forall \sigma \in L(N, M), \forall s \text{ service de } N, \exists \alpha \in (T - T_{ac})^* : \\ \sigma.\alpha \in L(N, M) \text{ et } \underline{\sigma}(s_{ac}) = \underline{\sigma.\alpha}(s_{ret})$$

Pour prendre une analogie, un entrepreneur de travaux publics qui commencerait la construction de votre maison tout en sachant qu'il n'a pas les fonds nécessaires pour la terminer (et qu'il lui faudra donc recevoir d'autres commandes pour pouvoir mener la vôtre à son terme) serait un serveur malhonnête.

Définition IX.3 - Critère de courtoisie

Un serveur sera dit **courtois** s'il ne peut fournir un résultat pour un service qu'après avoir au préalable accepté une requête pour ce service :

$$\forall \sigma \in L(N, M), \forall s \text{ service de } N, \underline{\sigma}(s_{ac}) \geq \underline{\sigma}(s_{ret})$$

En quelque sorte, le serveur doit répondre seulement s'il a été interrogé.

L'honnêteté et la courtoisie peuvent être vérifiées pour les RPO des ObCS sans la variable *id* (identificateur d'appel), car cette valeur n'est utilisée par le serveur que pour éviter de mélanger des jetons provenant de deux invocations différentes.

Pour exécuter l'étape 3 du procédé de construction du WSCS (§V.3), on a besoin d'un chemin entre la place paramètre d'un service et sa place résultat. L'existence d'un tel chemin est assurée par le critère de courtoisie.

$L(N, M) \mid T_{ac}$ peut être considéré comme le **contrat** qu'un serveur garantit de remplir pour l'ensemble de ses clients.

Considérons le cas d'un serveur dont le marquage est tel qu'il est en mesure d'accepter une requête, mais qui évolue par occurrence de transitions privées (TP) de telle sorte qu'il ne puisse plus accepter la requête initiale. Il ne remplit pas son contrat, car un client ne peut pas être informé de cette évolution interne, qui conduit cependant au rejet de sa requête.

Pour éviter de tels comportements, les ObCS d'implémentation et de spécification doivent être **transparents**

Définition IX.4 - Critère de transparence

Soit $\sigma \in L(N, M)$ et $s_{ac} \in T_{ac}$ tels que $\sigma.s_{ac} \mid T_{ac} \in L(N, M) \mid T_{ac}$.

Le serveur sera dit **transparent ssi** :

$$\exists \alpha \in (T - T_{ac})^* \text{ tel que } \sigma.\alpha.s_{ac} \in L(N, M)$$

Un serveur qui n'est pas transparent sera dit **sournois**.

La transparence est équivalente à la **condition de comportement** de [Andre 90], et elle garantit que le fait d'accepter une requête dépende uniquement des requêtes précédemment acceptées.

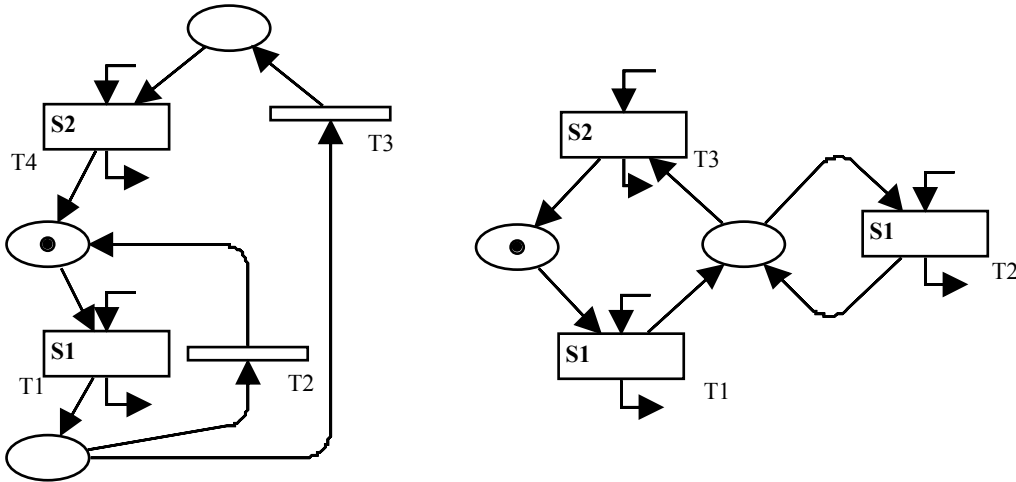


Figure IX.1 - Serveur sournois et serveur transparent

La Figure IX.1 montre, à gauche, un serveur sournois : après le franchissement de T1 suite à une requête de S1, le franchissement de T2 (resp. T3) empêche d'accepter le service S2 (resp. S1). Le réseau de droite, sur la même figure, modélise un serveur transparent qui accepte les mêmes séquences de requêtes de service.

3. COHERENCE ENTRE SPECIFICATION ET D'IMPLEMENTATION

Soient (N^{spec}, M^{spec}) et (N^{imp}, M^{imp}) les ObCS de spécification et d'implémentation d'un serveur. Il doivent remplir le même contrat pour les clients de ce serveur. Ceci est assuré par le critère suivant :

Définition IX.5 - Critère d'implémentation fidèle

Un ObCS d'implémentation est **fidèle** à un ObCS de spécification s'il est tel que :
il accepte une séquence de requête si et seulement si elle est également acceptée par l'ObCS de spécification, soit :

$$L(N^{spec}, M^{spec}) \mid T_{ac} = L(N^{imp}, M^{imp}) \mid T_{ac}.$$

Ainsi, du point de vue d'un client, il n'y a pas de différence entre les ObCS de spécification et d'implémentation.

4. OBSCS ET HERITAGE

Soient C^{gen} et C^{spec} deux classes d'objets telles que C^{spec} hérite de C^{gen} . La sémantique de l'héritage (relation est-un) impose qu'un client puisse utiliser une instance de C^{spec} exactement de la même manière qu'une instance de C^{gen} . Ceci est assuré par le critère suivant :

Définition IX.6 - Critère d'héritage du comportement

Un objet d'une classe C^{spec} dérivée d'une classe C^{gen} **hérite du comportement** de C^{gen} s'il accepte toute séquence d'appels de service acceptés par une instance de C^{gen} .

$$L(N^{sp}, M^{sp}) \mid T^{spec}_{ac} \supseteq L(N^{gen}, M^{gen}) \mid T^{gen}_{ac}.$$

Il faut remarquer que ce critère est très contraignant, car le plus souvent on a $T^{gen}_{ac} \subsetneq T^{spec}_{ac}$. Ceci correspond au fait qu'une instance de C^{spec} doit remplir le contrat d'une instance de C^{gen} en plus de son propre contrat.

Ce critère n'est à vérifier que pour les ObCS de spécification : s'il est vérifié pour l'ObCS de spécification, il l'est aussi pour l'ObCS d'implémentation, d'après le critère d'implémentation fidèle.

5. USAGE DE L'EQUIVALENCE SPECIFICATION/IMPLEMENTATION

Dans la plupart des systèmes, la relation d'utilisation entre objets est une hiérarchie comprenant un grand nombre de niveaux. Un petit nombre d'objets sont des clients purs, d'autres sont des serveurs purs, et les autres sont à la fois clients et serveurs.

Un des atouts majeurs d'un formalisme dédié à la conception structurée est de fournir la possibilité d'examiner le système à différents niveaux d'abstraction. Avec une approche descendante, un objet à la fois client et serveur est considéré uniquement comme un serveur, et les objets de plus bas niveau sont cachés. Avec une approche ascendante, un tel objet est considéré comme étant seulement un client, et les objets de plus haut niveau ne sont pas pris en compte.

Grâce à l'équivalence entre les ObCS de spécification et d'implémentation, le formalisme des Objets Coopératifs offre au concepteur la possibilité intéressante d'obtenir des modèles exécutables à chaque niveau d'abstraction.

Le procédé de construction du WSCS peut être appliqué à quelques classes d'objet, afin d'obtenir un modèle de fonctionnement d'une partie du système :

- Si on supprime d'un ObCS ses arcs d'activation et de complétion, l'objet n'est plus un serveur et le WSCS qui en résulte modélise un sous-système;
- Si au contraire on considère uniquement l'ObCS de spécification, l'objet n'est plus un client, et le WSCS qui en résulte est un modèle abstrait du système.

Ces deux techniques peuvent être utilisées simultanément, afin d'obtenir des modèles abstraits de sous-systèmes.

6. ANALYSE DES COMMUNICATIONS STATIQUES

Une transition d'invocation décrit une communication statique si toutes ses occurrences concernent le même serveur. C'est le cas, par exemple, si le serveur est désigné par une référence qui n'est jamais modifiée.

Quand toutes les communications d'un système sont statiques, le procédé de construction du WSCS peut être simplifié, puisqu'il n'est plus nécessaire de replier les ObCS des instances afin de produire les ObCS d'extension, ni de fusionner les places paramètre et résultat le long de la hiérarchie d'héritage.

Dans le cas d'un système à communications statiques, il est de plus possible de vérifier si les clients utilisent leurs serveurs correctement (i.e. en accord avec leur mode d'emploi) afin que ni les clients ni les serveurs n'atteignent un état de blocage.

Considérons un objet client O^C avec un ensemble T_{inv} de transitions d'invocation décrivant des communications statiques avec un serveur O^S .

Soient N^C et N^S les ObCS de O^C et O^S respectivement, étendus comme défini aux étapes 2 et 3 du procédé de construction du WSCS afin de permettre les communications, et soient T^C et T^S l'ensemble des transitions de N^C et N^S .

Si $T_{ac} \subseteq T^S$ est l'ensemble des transitions d'accord de N^S , il est possible de définir une fonction :

$$Lk : T_{inv} \rightarrow T_{ac}$$

où $Lk(t_{inv})$ est la transition de T_{ac} qui accepte le service invoqué par t_{req} .

Définition IX.7 - Compatibilité entre client et serveur

On dira que N^S est **compatible** avec N^C ssi le langage des services demandés à N^S par N^C est inclus dans le langage des services accepté par N^S , i.e. :

$$L(N^S, M^S) \mid T_{ac} \supseteq Lk(L(N^C, M^C) \mid T_{inv}).$$

Sous cette condition, O^S ne pose aucun problème pour O^C : N^C a le même comportement que les transitions de T_{inv} soient considérées comme des transitions ordinaires, ou que les communications aient effectivement lieu [Sibertin 91b].

Plus précisément, si N est le réseau résultant de la fusion des places paramètre et résultat de N^C et N^S suivant L_k , et si M est le marquage de N déduit de M^C et M^S , on a :

$$L(N, M) \mid T^C = L(N^C, M^C).$$

Ce résultat peut être utilisé incrémentalement si un client utilise plusieurs serveurs (il est compatible avec l'ensemble des serveurs s'il est compatible avec chacun individuellement), et si un serveur est lui-même client d'un autre serveur O^{SS} (si O^{SS} est compatible avec O^S et O^S est compatible avec O^C , alors le réseau qui résulte de la fusion des places paramètres et résultat de N^{SS} et N^S est compatible avec N^C).

Par contre, si un serveur est utilisé par plusieurs clients, $L(N^S, M^S) \mid T_{ac}$ est à comparer avec le langage de requêtes de l'union de ses clients : l'ajout d'un nouveau client à un serveur remet en question la compatibilité de celui-ci avec ses clients précédents.

Conclusion

Le formalisme des Objets Coopératifs, décrit dans ce mémoire, permet de structurer le modèle d'un système selon les concepts de l'approche objet, et de décrire sa dynamique en utilisant la théorie des Réseaux de Petri.

Les principales caractéristiques du formalisme sont les suivantes :

- L'unité de modélisation est l'**Objet**, entité qui regroupe des données, des opérations sur ces données et la structure de contrôle qui régit l'exécution de ces opérations ;
- Le formalisme est à **base de classe** : chaque objet est instance d'une classe qui définit sa structure et son comportement ;
- l'**Encapsulation** des données et du comportement des objets est assurée par la définition pour chaque classe d'une spécification (description destinée aux utilisateurs des instances de la classe) et d'une implémentation (description du fonctionnement exact d'une instance). Des critères formels de compatibilité entre spécification et implémentation sont définis ;
- Le formalisme autorise l'**instanciation dynamique** de nouveaux objets pendant l'activité du système ;
- Les classes d'objets sont structurées en une **hiérarchie d'héritage**, qui permet d'utiliser le principe du **polymorphisme** ;
- On peut définir des **classes composites**, qui décrivent des sous-systèmes d'objets fortement couplés et qui permettent de s'abstraire de la complexité locale d'un groupe d'objet en relation étroite ;
- Les objets d'un système communiquent par **invocation**, et le formalisme permet la liaison tardive, qui implémente le polymorphisme ;
- Le formalisme permet une explicitation complète et formelle de toutes les communications entre objets. Il permet de plus de distinguer les **flots de données** des flots de **contrôle** ;
- Il est aisé d'exprimer la **concurrency** au sein d'un système, aussi bien entre les objets qu'au sein de chaque objet. Plusieurs flots de contrôle peuvent évoluer concurremment au sein d'un système, et chaque objet peut être traversé par plusieurs flots de contrôle à la fois ;
- Le formalisme autorise la prise en compte du **temps** aussi bien en définissant des liens de causalité entre événements qu'en spécifiant des dates et des durées ;
- L'**environnement** d'un système peut être modélisé de manière uniforme, aussi bien comme un client que comme un serveur du système modélisé. Cette uniformité de description permet de déplacer à volonté la frontière entre le système modélisé et son environnement. De même, objets, sous-systèmes et systèmes peuvent être modélisés de la même façon ;
- La sémantique du formalisme est **définie formellement**, (ce qui n'est pas le cas pour tous les langages à objets et moins encore pour les LOO concurrents) en utilisant les Réseaux de Petri, l'une des trois théories mathématiques reconnues pour l'expression de la concurrence ;

- Le formalisme peut être **exécuté** (par semi-compilation ou interprétation) de manière répartie ;
- Enfin l'utilisation des RdP permet d'effectuer des **analyses** statiques des propriétés des modèles produits.

Formalisme de modélisation, ou langage de programmation ?

Doit-on situer les Objets Coopératifs comme un formalisme de modélisation des systèmes (pour en spécifier et/ou en concevoir le fonctionnement) ou comme un langage de programmation (pour en implanter les fonctionnalités) ?

En temps que formalisme de modélisation, les Objets Coopératifs prennent en compte les aspects fonctionnels (ce que fait un système) et structurels (comment il est fait). Le formalisme traite notamment les aspects liés au comportement, à la communication entre les composants et avec l'environnement, au parallélisme, au séquençement et à la durée des opérations.

- Les Objets Coopératifs disposent des mécanismes indispensables pour permettre l'intelligibilité du modèle d'un système complexe, et pour rendre rigoureux le processus de modélisation ;
- Le formalisme permet d'adopter un point de vue plus ou moins abstrait sur le système, en combinant les possibilités offertes par l'héritage, la distinction spécification/implémentation, et la relation de composition.

En temps que langage de programmation, les Objets Coopératifs ont les caractéristiques d'exécutabilité et de sémantique bien définie qui sont indispensables pour une utilisation en tant qu'outil d'implantation.

- L'exécution par interprétation ne pose pas de problème, et les outils qui permettraient une telle exécution sont déjà disponibles (interprètes de réseaux de haut niveau) ;
- L'exécution par compilation (totale, en générant à partir d'un modèle les structures de données et de contrôle d'un langage concurrent, ou partielle, en utilisant les techniques empruntées à l'intelligence artificielle) pose certainement des problèmes plus complexes, et il faudrait sans doute dans ce cas se restreindre à des réseaux bornés.

Evolution et enrichissement du formalisme

Le formalisme des Objets Coopératifs se situe à un haut niveau d'abstraction, et il est utilisable pour une classe très large de problèmes, selon la façon dont les concepts ou constructions qu'il offre sont interprétés. Des études de cas on d'ores et déjà été réalisées dans le domaine des ateliers de production, des Systèmes d'Information, des interfaces personne/logiciel et de la circulation de documents.

L'utilisation dans d'autres domaines suggérera sans doute de nouveaux enrichissements ; par exemple c'est la modélisation des ateliers flexibles qui a suggéré la définition des places triées et la hiérarchie de composition, extensions qui ne sont pas forcément significatives dans d'autre domaines.

L'utilisation de ce formalisme en tant que langage de programmation nécessiterait certainement d'autres mécanismes (on peut penser par exemple à un mécanisme d'exceptions permettant de prendre en compte les cas de mort des objets).

Objets Coopératifs et méthodologie

Les Objets Coopératifs présentent le grand avantage de pouvoir être utilisés (en tant que formalisme ou en temps que langage) tout au long du cycle d'un développement logiciel.

Toutefois, si un formalisme peut être d'usage général, une **méthode** est nécessairement très liée à son domaine d'utilisation, et il faudrait développer, pour chaque domaine, un corpus de règles méthodologiques permettant l'utilisation correcte du formalisme.

Le formalisme des Objets Coopératifs est en lui-même l'apport central de ce mémoire. Nous souhaitons également mettre en avant deux autres contributions originales :

- Nous avons défini au chapitre II.2 des extensions aux Réseaux de Petri à Objets (notamment les places triées, les arcs inhibiteurs généralisés et le registre temporel), qui sont sans nul doute applicables à la plupart des dialectes de RdP de haut niveau ;
- Nous avons prouvé au chapitre V, en construisant un réseau prédicat/transition (le WSCS) qui modélise le comportement d'un système d'Objets Coopératifs, que les RdP de haut niveau sont suffisamment puissants pour décrire tous les concepts de l'approche objet, y compris l'instanciation, l'accès chaîné aux attributs, la relation d'utilisation dynamique, l'héritage et le polymorphisme, ce qui à notre connaissance n'avait jamais été montré.

Durant toute la conception de ce formalisme, nous avons eu le souci d'en préserver la simplicité et l'utilisabilité. Nous avons eu la chance de rencontrer des personnes suffisamment motivées pour bien vouloir évaluer l'adéquation du formalisme à différents domaines, et ces expérimentations se sont révélées riches d'enseignements.

Nous sommes bien entendu conscients qu'une utilisation du formalisme à plus grande échelle ne peut être envisagée sans disposer d'un environnement support, qui faciliterait la saisie des modèles, leur vérification syntaxique et sémantique, et qui supporterait les activités de test, de simulation et de prototypage. C'est dans cette voie que nous souhaitons nous diriger.

Glossaire

Accusé d'exécution

Un *client* est toujours informé de l'achèvement d'une *invocation* qu'il a adressée à un *serveur*. Si le *service* comporte des paramètres en sortie, le client est informé par la disponibilité des valeurs résultats ; Sinon, il l'est par un signal appelé Accusé d'exécution.

Activité spontanée

Activité d'un objet qui n'est pas la conséquence d'une invocation de *service*, mais qui est entreprise de la propre initiative de l'objet, en fonction de son état. L'activité spontanée d'un objet est décrite dans l'*ObCS d'implémentation* de sa classe.

Arc d'activation

Arc pendant en entrée d'une *transition de service*, étiqueté par les paramètres formels d'entrée du service. Lors de la construction du WSCS, cet arc relie la *place paramètre* à la *transition d'accord* du service.

Arc de complétion

Arc pendant en sortie d'une transition de service, étiqueté par les paramètres formels de sortie du service. Lors de la construction du WSCS, cet arc relie la *place résultat* à la *transition de complétion* du service.

Attribut

Elément de la structure de donnée d'une classe, et défini dans son implémentation par un nom et un *type*. Si son type est un *type classe* l'attribut est une *référence*, si c'est un *type simple*, l'attribut est une *propriété*.

Classe atomique

Ou Classe d'objets atomiques. Classe feuille du graphe de la relation de composition. Ses instances ne sont pas décomposables en instances de classes plus simples, par opposition à *classe composite*.

Classe composite

Ou Classe d'objets composites. Classe dont les instances sont composées d'instances de classes atomiques ou composites, par opposition à *classe atomique*.

Client

Objet auteur d'une invocation. Pour cette invocation, il fournit le nom de l'objet *serveur* et les paramètres d'entrée du *service*, et en récupère les valeurs de retour. Par extension *Classe cliente* : Classe dont l'implémentation contient des *TI*.

Client pur

Objet dont la classe ne publie pas de *service*. Par extension *Classe de clients purs* : racine du graphe de la relation d'utilisation.

Comportement

Elément de la description d'une classe d'objets, qui spécifie son activité et son évolution. Le comportement d'un objet est décrit par l'*ObCS* (de spécification ou d'implémentation) de sa classe.

Coopération

Elément de la description d'une classe d'objets, qui spécifie la nature des relations qu'un objet entretient avec d'autres objets du système. La coopération d'un objet est décrite par l'*ObCS d'implémentation* de sa classe.

Create (opération)

Opération décrivant comment les instances d'une classe doivent être créées et initialisées. Toute classe publie une opération *create*, qui définit le marquage initial de l'*ObCS* de ses instances.

Estampille d'appel

Elément d'information généré lors de l'invocation d'un service, permettant l'acheminement correct des résultats vers le client de l'appel.

Garde

Valeur entière spécifiant le nombre d'unités de temps pendant lequel un *client* est disposé à attendre la prise en compte d'une *invocation* qu'il a adressée à un *serveur*.

Invocation

Action consistant en l'appel, par un objet *client* d'un service publié par la classe d'un objet *serveur*. Le client fournit les paramètres d'entrée du service. L'objet serveur d'une invocation est dénoté, dans l'ObCS du client, par une variable d'entrée de la *transition d'invocation* ou par une *référence* du client.

Invocation non confirmée

Mode d'invocation dans lequel le flot de contrôle qui a déclenché une *invocation* chez un *client* continue sans attendre ni l'exécution, ni la prise en compte de l'invocation par le *serveur*.

ObCS

Object Control Structure ou structure de contrôle d'objet. L'ObCS d'une classe décrit le comportement des objets instances de cette classe, et précise son mode d'emploi. Pour une classe, deux ObCS sont fournis : l'*ObCS de spécification* et l'*ObCS d'implémentation*. Un ObCS est décrit par un Réseau de Petri à Objets, et par la correspondance entre les services offerts par l'objet et des transitions de ce réseau.

ObCS d'extension

Etape intermédiaire de la construction du WSCS, l'ObCS d'extension d'une classe est un RPO pouvant faire évoluer un nombre quelconque d'instances de cette classe.

ObCS d'implémentation

Décrit le fonctionnement interne d'un objet, en précisant notamment son *activité spontanée* et sa *Coopération* avec d'autres objets du système.

ObCS de spécification

Décrit le comportement externe d'un objet, tel qu'il est perçu par ses *clients*. Il spécifie le mode d'emploi de l'objet, les séquences exécutables d'activation de ses services.

Objet actif

Objet dont la classe possède un ObCS qui contraint l'activation des services qu'il offre.

Objet passif

Objet dont la classe ne publie que des *Data-Flows*. Un objet passif possède un ObCS trivial.

Objet primitif

Objet qui préexiste à l'initialisation d'un *Système d'Objets Coopératifs*.

Place d'attente

Place introduite lors de la construction du WSCS. Pour chaque *TI*, une place d'attente se trouve en sortie de la *transition d'appel* et en entrée de la *transition de complétion*. Elle est marquée si une invocation du service est en cours. Son marquage est la concaténation des variables d'entrée de la *TI* avec l'*estampille d'appel*.

Place paramètre

Place introduite lors de la construction du WSCS. Pour chaque *TI*, une place paramètre se trouve en sortie de la *transition d'appel* et en entrée de la *transition d'accord*. Elle contient les paramètres de l'invocation et l'*estampille d'appel*.

Place résultat

Place introduite lors de la construction du WSCS. Pour chaque *TI*, une place résultat se trouve en sortie de la *transition de retour* et en entrée de la *transition de complétion*. Elle contient les résultats de l'invocation et l'*estampille d'appel*.

Propriété	<i>Attribut de type simple</i> ou de <i>type construit</i> . Les propriétés sont manipulées par valeur. Voir aussi : <i>référence</i> .
Référence	<i>Attribut de type classe</i> . La valeur d'une référence est le nom d'un objet de cette classe.
Rendez-vous	Mode d'appel par défaut lors de l'invocation. Du côté <i>client</i> , le flot de contrôle qui a conduit à l'invocation est suspendu, jusqu'à ce que le service ait été mené à son terme par le <i>serveur</i> .
Rendez-vous gardé	Mode d'appel où le client définit une <i>garde</i> spécifiant son délai maximum d'attente de la prise en compte de son <i>invocation</i> par le <i>serveur</i> .
Serveur	Objet qui reçoit une invocation concernant l'un des services publiés par sa classe.
Serveur pur	Objet dont l' <i>ObCS d'implémentation</i> ne contient pas de <i>TI</i> . Par extension <i>Classe de serveurs purs</i> : feuille du graphe de la relation d'utilisation.
Service	Opérateur publié par une classe, permettant de faire évoluer l'état d'une instance ou d'obtenir une information sur cet état. Un service est défini par son nom et sa <i>signature</i> .
Service d'initialisation	<i>Service</i> permettant de fixer la valeur initiale d'un <i>attribut</i> des instances d'une classe.
Service d'interface	<i>Service</i> pouvant être invoqué par l'environnement d'un système d'Objets Coopératifs.
Service de mise à jour	<i>Service</i> permettant de faire évoluer la valeur d'un <i>attribut</i> de l'instance.
Signature (d'un service)	Liste du nom et du type des ses paramètres d'entrée et de ses valeurs de retour.
TA	Transition d'Accès, transition qui contient l'accès à un attribut publié par la classe d'un autre objet, de la forme <i>variable.attribut</i> , soit dans sa précondition, soit dans son action, soit dans le code d'une <i>opération interne</i> qu'elle appelle.
TI	Transition d'Invocation, transition dont l'action est une invocation de la forme <i><résultat> :=variable.service<liste de paramètres></i> .
Transition d'Accès	voir <i>TA</i>
Transition d'accord	Dans l' <i>ObCS d'implémentation</i> , transition dont la franchissabilité définit sous quelle condition une requête de service peut être acceptée par l'objet, en fonction du marquage de son <i>ObCS</i> . Peut être confondue avec la <i>transition de retour</i> .
Transition d'appel	Transition introduite lors de la construction du <i>WSCS</i> . Toute <i>TI</i> est éclatée en une transition d'appel et une <i>transition de complétion</i> , connectées par une <i>place d'attente</i> .
Transition d'instanciation	Transition introduite lors de la construction du <i>WSCS</i> pour modéliser l'opération <i>create</i> .
Transition d'Invocation	voir <i>TI</i>

Transition de complétion

Transition introduite lors de la construction du WSCS. Toute *TI* est éclatée en une *transition d'appel* et une transition de complétion, connectées par une *place d'attente*.

Transition de retour

Dans l'ObCS d'implémentation, transition qui produit dans l'un de ses arcs de sortie les valeurs de retour d'un service. Peut être confondue avec la *Transition d'accord*.

Transition de service

Dans l'ObCS de spécification, transition associée à un service publié par la classe et qui modélise ses conditions d'activabilité.

Transition Privée

Transition qui n'est pas associée à l'exécution d'un service et qui modélise une évolution interne de l'état d'un objet.

Type classe

Type dont le domaine de valeur est l'ensemble des nom des objets de cette classe. Les variables de type classe sont appelées *références*.

Type construit

Type simple construit à partir d'autres types par application d'un constructeur de type.

Type prédéfini

Un des *types simples* suivants : INTEGER, REAL, CHARACTER, BOOLEAN, STRING.

Type simple

Type prédéfini ou type construit, dont les instances sont manipulées par valeur.

WSCS

Whole System Control Structure : C'est un réseau de Petri à Objets décrivant la structure de contrôle d'un système d'Objets Coopératifs, construit à partir des *ObCS* des classes qui composent ce système.

Bibliographie

- [Agha 86a] AGHA G.
Actors: A Model of Concurrent Computation in Distributed Systems.
Series in Artificial Intelligence.
Cambridge (Ma) USA, MIT Press.
- [Agha 86b] AGHA G.
An Overview of Actor Languages.
SIGPLAN Notices, Vol. 21, n° 10, October 1986
- [Agha 90] AGHA G.
Concurrent Object-Oriented Programming.
Communications of the ACM, Vol. 33, n° 9, pp.125-141
September 1990
- [Alla 87] ALLA H.L.
Réseaux de Petri colorés et Réseaux de Petri continus : application à l'étude des systèmes à évènements discrets.
Thèse de doctorat d'état de l'institut national polytechnique de Grenoble.
Grenoble, Juin 1987
- [America 87] AMERICA P.
Inheritance and Subtyping in a Parallel Object-Oriented Language
in [ECOOP 87] pp. 281-290
- [Andre 82] ANDRE C.
Use of the Behaviour equivalence in Place-Transition Net Analysis;
in Application and Theory of Petri Nets, IFB 52, Springer, 1982
- [André 81] ANDRE C.
Systèmes à évolutions parallèles : modélisation par réseaux de Petri à capacité et analyse par abstraction.
Thèse d'état, Université de Nice, Février 1981
- [Audureau...87] AUDUREAU E. ; FARIÑAS DEL CERRO L. ; ENJALBERT P.
Théorie de la programmation et logique temporelle. Première partie : Validation d'algorithmes séquentiels.
Technique et Science Informatique, Vol.6 n° 6. pp.527-540, 1987
- [Audureau...88] AUDUREAU E. ; FARIÑAS DEL CERRO L. ; ENJALBERT P.
Théorie de la programmation et logique temporelle. Deuxième partie : Validation d'algorithmes parallèles.
Technique et Science Informatique, Vol.7 n° 2. pp.181-200, 1988
- [Bako 90] BAKO B.
Mise en œuvre et simulation du niveau coordination de la commande des ateliers flexibles : une approche mixte réseaux de Petri et systèmes de règles.
Thèse de L'université Paul Sabatier de Toulouse
Toulouse, Octobre 1990
- [Barron 82] BARRON J.
Dialogue and Processing Design for Interactive Information Systemes using TAXIS.
ACM conference on Office Information Systems, Philadelphia (Pen), June 1982
- [Barthet...86] BARTHET M.F. ; SIBERTIN-BLANC C.
La modélisation d'applications interactives appliquées aux utilisateurs par des réseaux de Petri à structures de donnée.
Actes du 3° colloque-exposition de Génie Logiciel, pp 117-136
Versailles, mai 1986

-
- [Bastide...90] BASTIDE R., PALANQUE P.
Petri Net Objects for the design, validation and prototyping of user-driven interfaces.
Proceedings of IFIP conference Interact'90 Cambridge August 1990. North-Holland.
- [Bastide...91a] BASTIDE R, SIBERTIN-BLANC C.
Modelling a flexible manufacturing system by means of Cooperative Objects.
Proceedings of IFIP conference CAPE'91 on computer applications in production and engineering.
September 1991, Bordeaux, France.
- [Benoit...86] BENOIT CH. ; CASSEAU Y. ; PHERIVONG CH.
LORE : Un langage Objet Relationnel et Ensembliste.
in [BI-GL 48] pp 3-13
- [Berthelot 86] BERTHELOT G.
Transformations and decompositions of Nets. In [Brauer...86]
- [Bezivin 84] BEZIVIN J.
Simulation et Langages Orientés-Objet.
Secondes Journées AFCET sur les langages Orientés-Objet.
BIGRE+GLOBULE N°41, pp 194-211, 1984
- [BI-GL 48] BIGRE+GLOBULE N° 48 Janvier 1986
Actes des journées AFCET-Informatique Langages Orientés Objet.
IRCAM, Janvier 1986.
- [BI-GL 49] BIGRE+GLOBULE N° 49 Juin 1986
Actes des journées ADA AFCET-ENST
Paris, Juin 1986.
- [BI-GL 57] BIGRE+GLOBULE N° 57 Décembre 1987
Actes des journées ADA AFCET-ENST "Le parallélisme en ADA"
Paris, Décembre 1987.
- [Bjørner...90] BJØRNER D. ; DRUFFEL L.
Position Statement: ICSE-12 Workshop on Industrial Experience using Formal Methods.
in [ICSE'12] pp. 264-266
- [Blake 87] BLAKE E. ; COOK S.
On Including Part Hierarchies in Object-Oriented Languages, with an Implementation in Smalltalk.
in [ECOOP 87] pp. 45-56
- [Bleser...82] BLESER T. ; FOLEY J.D.
Towards Specifying and Evaluating Human Factors of User-Computer Interfaces.
Human Factors in Computer Systems Proceedings, march 1982
- [Bobrow...81] BOBROW D.G. ; STEFIK M.
The LOOPS Manual.
Technical report KB-VLSI-81-13
Xerox Palo Alto Research Center, Palo Alto, Cal, 1981
- [Booch 87] BOOCH G.
Software Engineering with Ada. Benjamin/Cummings Publishing, California, 87
- [Borning...81] BORNING A.H. ; INGALLS D.H.H.
Multiple Inheritance in Smalltalk-80.
Proc. of the AAAI Conference, Pittsburg, August 1982, pp234-237
- [Brachman 83] BRACHMAN R.J.
What IS-A is and isn't: An analysis of taxonomic links in semantic networks.
IEEE Computer, vol.16, n° 10, pp.67,73, Oct 1983

-
- [Brams 83] BRAMS G.W. (nom collectif)
Réseaux de Petri : Théorie et Pratique. T.1 : théorie et analyse ; T.2 modélisation et applications.
ISBN 2-903-60713-3 Masson 1983
- [Brauer...86] BRAUER W.
Petri nets: Applications and relationships to other models of concurrency.
W. Brauer, W. Reisig, G. Rosenberg editor, LNCS 254 &255, Springer Verlag.
- [Brinch Hansen 72] BRINCH HANSEN P. Structured Multiprogramming.
Communications of the A.C.M. Vol 15, N° 7, pp 574-578, 1972
- [Brinch Hansen 73] BRINCH HANSEN P. Operating Systems Principles.
Prentice-Hall 1973
- [Briot...87] BRIOT J.P. ; YONEZAWA A.
Inheritance and Synchronization in Concurrent OOP.
in [ECOOP 87] pp. 35-44
- [Bruno...86] BRUNO G., ALESSANDRO B.
Petri-net based Object-Oriented modelling of distributed systems. OOPSLA'86, 1986
- [Buetler...89] BUETLER B., ESSER R., MATTMAN R.
A distributed simulator for High Order Petri nets.
10th European Workshop on applications and theory of Petri Nets, Bonn, June 89.
- [Buhr 84] BUHR R.J.A.
System Design with Ada.
Prentice Hall 1984
- [Cardelli 84] CARDELLI L.
A Semantics of Multiple Inheritance.
in "Semantics of Data Types", LNCS N° 173, pp 51-67.
Springer-Verlag, New-York 1984
- [Cardelli...83] CARDELLI L. ; WEGNER L.
On Understanding Types, Data Abstraction and Polymorphism.
Computing Surveys, vol.17, n° 4, pp 471-522, Dec. 1985
- [Caromel 90] CAROMEL D.
Concurrency: An Object-Oriented Approach.
in [TOOLS'90] pp. 183-197
- [CNRS 88] GROUPE DE REFLEXION TEMPS REEL du CNRS
Rapport "Le temps réel"
Département Sciences Physiques pour l'Ingénieur du CNRS, Juin 1988
- [Cointe 86] COINTE P.
Une introduction à la programmation par objet.
Giens 86 : 2° journées Bases de Données avancées.
- [Colom...86] COLOM J. M., SILVA M., VILLAROEL J. L.
On software implementation of Petri Nets and Colored Petri Nets using high-level concurrent languages.
Seventh European Workshop on applications and theory of Petri nets, Oxford, July 86.
- [Coutaz 87] COUTAZ J.
The Construction of User Interfaces and the Object Paradigm.
in [ECOOP 87] pp. 135-144
- [Cowan...90] COWAN B.W. ; WEIN M.
State versus history in user interfaces
Proceedings of IFIP conference Interact'90 Cambridge August 1990. North-Holland.

-
- [Cox 86] COX B.J.
Object-oriented programming : an evolutionary approach.
Addison-Wesley, Mass., 1986
- [Dahl...66] DAHL O.J ; NYGAARD K.
SIMULA - An Algol-Base Simulation Language
Communications of the ACM, Vol. 9, N°9, pp. 671-678
Sept. 1966
- [Dang 86] DANG W.
Programmation Concurrente en Langage Orienté Objet.
in [BI-GL 48] pp. 149-155
- [De Bondeli 87] DE BONDELI P.
Implémentation en Ada de types de synchronisation et communication inter-tâches
spécifiés par des réseaux prédicats-transition.
in [BI-GL 57]
- [De Champeaux 91a] DE CHAMPEAUX D.
Object-Oriented Analysis and Top-Down Software Development.
ECOOP'91 LNCS N°512, pp 360-376
Springer-Verlag July 1991
- [De Champeaux 91b] DE CHAMPEAUX D.
A Comparative Study of Object-Oriented Analysis Methods.
HP-Labs. technical report.
April 1991
- [DeCindio...81] DE CINDIO F., DE MICHELIS G., POMELLO L, SIMONE
Superposed Automata nets. Applications and Theory of Petri nets, LNCS 52, 1981.
- [Denert 77] DENERT E.
Specification and design of dialogue systems with state diagrams.
in Proc. of International Computing Symposium, Liège, pp. 417-424
North-Holland 1977.
- [Dijkstra 65] DIJKSTRA E.W.
Cooperating sequential processes.
Technical Report EWD 123, 1965,
reproduit dans Programming Languages, F.Genuys (ed.) Academic Press, 1968, pp. 43-
112
- [Dijkstra 71] DIJKSTRA E.W.
Hierarchical ordering of sequential processes.
Acta Informatica, Vol.1, N° 2, pp. 115-138, 1971
- [Ducourneau...87] DUCOURNEAU R. ; HABIB M.
On some Algorithms for Multiple Inheritance in Object Oriented Programming.
in [ECOOP 87] pp. 291-302
- [Dumond 89] DUMOND Y.
Démarches applicatives par les objets dans la conception de logiciel de commande de
procédés industriels.
in [GL17 89] pp.68-83
- [ECOOP 87] EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING
Special issue of BIGRE N° 54, Juin 1987
- [Falk 88] FALK H.
CASE Tools emerge to handle real-time systems.
Computer Design Vol.27 N° 1 January 1988 pp 53-74

-
- [Feldbrugge...87] FELDBRUGGE F. ; JENSEN K.
Petri Net Tool Overview
in [LNCS 87], pp. 20-61
- [Ferber 84] FERBER J.
Quelques aspects du caractère self-réflexif du langage MERING.
BIGRE N° 41
- [Fritschy 89] D. FRITSCHY
Cahier des charges pour l'acquisition d'un outil de simulation temporelle des flux de production;
SORMEL, Besançon , 1989
- [Galton 87] GALTON A (Editor)
Temporal Logics and their Applications.
Academic Press 1987
- [Genrich 87] Predicate / Transition Nets
in [LNCS 87] Part I Vol.254 pp. 207-247
- [GL17 89] GENIE LOGICIEL ET SYSTEMES EXPERTS
N° 17 Décembre 1989 "Ada-Les technologies «Objets»".
ISSN 0295-6322
- [Goldberg...83] GOLDBERG A. ; ROBSON D.
Smalltalk-80, The Language and its Implementation.
Addison-Wesley, 1983
- [Guttag 77] GUTTAG J.
Abstract Data Types and the Development of Data Structures.
Communications of the ACM, Vol.20, n° 6, June 1977
- [Hack 72] HACK M.H.T.
Analysis of production schemata by Petri Nets. TR-94, MIT, Boston 1972.
- [Hack 75] HACK M.
Petri Nets Languages
Comput. struct. Gr. Memo 124.
Project HAC. ; M.I.T. Cambridge Mass. (Juin 1975)
- [Halbert...86] HALBERT D.C. ; O'BRIEN P.D.
Using Types and Inheritance in Object-Oriented Languages.
in [ECOOP 87] pp. 23-34
- [Harel...88] HAREL D., LACHOVER H., NAAMAD A., PNUELI A., POLITI M., SHERMAN R.,
STHUL-TRAURING A
STATEMATE: A Working environment for the development of Complex Reactive Systems.
10th IEEE International conference on Software Engineering, Singapore, April 1988.
- [Hee...89] VAN HEE K.M. ; SOMERS L.J. ; VOORHOEVE M.
Executable Specifications for Distributed Information Systems.
in Informations Systems Concepts: An In-depth Analysis.
E.D. Falkenberg and P. Lindgreen (Editors)
Elsevier Science Publishers B.V. (North-Holland) 1989
- [Hewitt 77] HEWITT K.
Viewing control as patterns of message passing
AI Magazine 3(8) 1977.
- [Hewitt...73] HEWITT C. et al.
A Universal Modular Actor Formalism for Artificial Intelligence.
Proc of Int. Jnt. Conference on Artificial Intelligence, 1973

-
- [Hewitt...77] HEWITT C. ; BAKER H.
Laws for Parallel Communicating Processes.
IFIP-77 Toronto 1977
- [Hoare 69] HOARE C. A. R.
An axiomatic basis for computer programming. Communication of the ACM 12,10,
1969
- [Hoare 74] HOARE C.A.R.
Monitors : An Operating System Structuring Concepts.
Communications of the A.C.M. Vol 17, N° 10, pp 549-557, 1974
- [Hoare 78] HOARE C.A.R.
Communicating Sequential processes.
Communications of the A.C.M. Vol. 21, N° 8, pp 666-677, 1978
- [Hollinger 85] HOLLINGER D.
Utilisation pratique des réseaux de Petri dans la conception des systèmes de production.
TSI 0752-4072 1985
- [Hollinger 86] HOLLINGER D.
Réseaux de Petri et Flavors pour la spécification et la simulation de systèmes
productiques.
in [BI-GL 48] pp. 206-215
- [HOOD 89] European Space Agency
HOOD Reference Manual, issue 3.0; ESA, ref. WME/89-173/JB; Sept. 1989
- [Horn 87] HORN C.
Conformance, Genericity, Inheritance and Enhancement.
in [ECOOP 87] pp. 269-280
- [Howard 76] HOWARD J.H.
Proving Monitors.
Communications of the A.C.M. Vol 19, N° 5, pp 273-279, 1976
- [Huber...89] HUBER P., JENSEN K., SHAPIRO R. M.
Hierarchies in Coloured Petri nets.
10th European Workshop on applications and theory of Petri Nets, Bonn, June 89.
- [Hur...87] HUR J. ; CHON K.
Overview of a Parallel Object Oriented Language CLIX.
in [ECOOP 87] pp 315-324
- [IBM 89] IBM Corp.
Common User Access : Advanced Interface Design Guide.
June 1989
- [ICSE'12] Proceedings of 12th International Conference on Software Engineering.
Nice, France, March 1990.
IEEE Computer Society Press, ISSN 0270-5257
- [Jantzen...79] JANTZEN M. ; VALK R.
Formal properties of place/transitions nets.
In "Net Theory and Applications" Proceedings of the Advanced Course on General Net
Theory of Processes and Systems. Hambourg, RFA, 1979
- [Järvinen 90] JÄRVINEN H.M. ; KURKI-SUONIO R. ; SAKKINEN M. ; SYSTÄ K.
Object-Oriented Specification of Reactive Systems.
in [ICSE'12] pp. 63-71

-
- [Jensen 81] JENSEN K.
Coloured Petri nets and the invariant method.
Theoretical Computer Science n°14, pp 317-336, 1981
North-Holland Publishing Company.
- [Jensen 87] Coloured Petri Nets
in [LNCS 87] Part I Vol.254 pp. 248-299
- [Kahn...86] KAHN K. ; DEAN TRIBBLE E. ; MILLER M.S. ; BOBROW D.G.
Objects in Concurrent Logic Programming Languages.
OOPSLA'86, Special issue of SIGPLAN Notices, Vol. 21, N° 11, pp 214-223
Portland OR, USA, November 1986
- [Kaplan...89] KAPLAN S.M. ; LOYALL J.P.
GARP/Scheme: Implementing a Concurrent, Object-Based Language.
Special issue of BIGRE N° 65 : "Putting SCHEME to work", pp 66-84
ISSN 0221-5225, Juillet 1989
- [Kessels 77] KESSELS J.L.W.
An alternative to event queues for synchronization in monitors.
Communications of the A.C.M. Vol. 20, N° 7, pp 500-503 1978
- [Kopetz...87] KOPETZ H. ; OCHSENREITER W.
Clock synchronisation in distributed real-time systems.
IEEE Transactions on Computers, Vol. C-36, n °8, August 1987
- [Krakowiak 85] KRAKOWIAK S.
Des systèmes d'exploitation des ordinateurs.
ISBN 2-04-016416-2, Dunod Informatique 1985
- [Lamport 83] LAMPORT L.
Solved Problems, Unsolved Problems and Non-Problems in Concurrency.
Proceedings of 3rd Annual ACM Symposium on Principles of Distributed Computing,
1984
- [Lapalme...89] LAPALME G. ; SALLE P.
Plasma II: an Actor Approach to Concurrent Programming.
Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent
Programming.
SIGPLAN Notices, Vol. 24, N° 4, April 1989, pp. 81-83
- [Lautenbach...74] LAUTENBACH K., SCHMID H.
Use of Petri Nets for proving correctness of concurrent Process system.
Information Processing 74, North-Holland, 1974.
- [Lieberman 81] LIEBERMAN H.
A preview of Act-1.
AI-memo 625, MIT AI Lab. 1981
- [Lieberman 86] LIEBERMAN H.
Delegation and Inheritance: Two Mechanisms for Sharing Knowledge in Object-
Oriented Systems.
in [BI-GL 48] pp. 79-89
- [Liskov...77] LISKOV B. ; SNYDER A. ; ATKINSON R. ; SCHAFFERT C.
Abstraction Mechanisms in CLU.
Communications of the ACM, Vol.20, n° 8, August 1977
- [LNCS 87] Lecture Notes in Computer Science.
N° 254 : Petri Nets : Central Models and Their Properties.
N° 255 : Petri Nets : Applications and Relationships to Other Models of Concurrency.
Advances in Petri Nets 1986, Part I & II ; Proceedings of an Advanced Course, Bad
Honnet, September 1986. Brauer, Reisig, Rozenberg (Eds) Springer 1987

-
- [Mayer 91] MAYER R.J.
Wholistic design, Engineering and Manufacture
IFIP conference CAPE'91 on computer applications in production and engineering.
September 1991, Bordeaux, France.
- [Merlin 74] MERLIN P.
A study of the recoverability of computer systems.
Ph.D. thesis, University of California.
Irvine, 1974
- [Merlin...76] MERLIN P. ; FARBER D.J.
Recoverability of communication protocols-Implications of a theoretical study.
IEEE transactions on Communications, vol. 24, No 9, Septembre 1976
- [Meyer 90a] MEYER B.
Sequential and Concurrent Object-Oriented Programming.
in [TOOLS'90] pp. 17-28
- [Meyer 90b] MEYER B.
Conception et Programmation par Objets.
InterEditions, Paris 1990
- [Milner 80] MILNER R.
A Calculus of Communicating Systems.
Theoretical Computer Science n°23, pp. 267-310, 1983
Springer 1980
- [Milner 83] MILNER R.
Calculi for Synchrony and Asynchrony.
LNCS Vol. 92
Springer 1980
- [Minkowitz...87] MINKOWITZ C. ; HENDERSON P.
An Application of Object-Oriented Programming to Petri Net Models of Discrete
Event-Driven Simulation.
in [ECOOP 87] pp. 187-192
- [Minsky 75] MINSKY M.
A Framework for Representing knowledge.
in "Psychology of Computer vision", Winston, P (Ed.)
Mc-Graw & Hill, New-York 1975
- [Minsky 88] MINSKY M.
La Société de L'esprit.
InterEditions 1988
- [Moss...87] MOSS J.E.B. ; KOHLER W.H.
Concurrency Features for the Trellis/Owl language.
in [ECOOP 87] pp. 223-232
- [Oberweis 86] RICHTER G.
Temporal Aspects in Office Automation Systems.
Proc. of IFIP TC8 Working Conference : Office Systems: Methods and Tools, Pisa,
Italy, October 1986.
Brachhi G. , Tsihrizis D. (Eds.)
Elsevier Science Publishers B.V. (North Holland) © IFIP 1986
- [Olsen...83] OLSEN D.R. ; DEMPSEY E.P.
Syngraph : A Graphical User Interface Generator.
Computer Graphics July 1983, pp 43-50

-
- [Oswald...90] OSWALD H. ; ESSER R. ; MATTMAN R.
An Environment for Specifying and Executing Hierarchical Petri Nets.
in [ICSE'12] pp. 164-172
- [Paludetto 91] PALUDETTO M.
Sur la commande de procédés industriels : Une méthodologie basée Objets et Réseaux de Petri.
Thèse de Doctorat de l'Université Paul Sabatier.
Toulouse, Décembre 1991
- [Paludetto...90] PALUDETTO M. ; VALETTE R. ; COURVOISIER M.
Génération de code Ada à partir d'une approche Orientée-Objets HOOD/ Réseaux de Petri.
Proceedings of 3rd International Workshop on Software Engineering and its Applications
EC2 Toulouse, France, December 1990
- [Parnas 69] PARNAS D.L.
On the use of transition diagrams in the design of a user interface for an interactive computer system.
in Proc. of 24th National ACM conference, San Francisco, Calif. 1969 pp. 379-385.
- [Parnas 73] PARNAS D.L.
A technique for the specification of software modules with examples.
Communications of the ACM vol. 15, N° 5, pp 330-336
Mai 1973
- [Pasquier...89] PASQUIER A. ; ESTEVE P. ; ROUBINE O. ; QUEINNEC C. ; DELACOUR V.
FLTR3 : Extensions à objets pour un langage temps réel fortement typé.
in [GL17 89] pp.58-67
- [Peterson 81] PETERSON J.L.
Petri net theory and the modeling of systems.
Prentice-hall. 1981.Isbn 0-13-661983-5.
- [Petri 62] PETRI C.
Kommunikation mit automaten
PhD Dissertation, University of Bonn, 1962
- [Pitette 86] PITETTE G.
Programmation orientée objet avec Ada : Possibilités, Difficultés, Expérience d'Enseignement.
in [BI-GL 49], pp. 92-102
- [Pnueli 86] PNUELI A.
Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends.
LNCS N° 224, pp. 510-584
Springer-Verlag 1986
- [Pomello 86] POMELLO L.
Some equivalence notions for concurrent systems, an Overview.
G. Rozenberg edit., Advances in Petri Nets' 86, LNCS 222, 1986
- [Ramchandani 74] RAMCHANDANI C.
Analysis of asynchronous concurrent systems by timed Petri net models.
PhD thesis, MIT, Project MAC TR-120 Février 1974
- [Reizig 86] REIZIG W.
Petri Nets in Software Engineering; in [Brauer...86]

-
- [Richter 85] RICHTER G.
Clocks and their use for time modelling.
Information Systems : Theoretical and Formal Aspects.
A. Sernadas, J. Bubenko, A. Olivé (Eds.)
Elsevier Science Publishers B.V. (North Holland) © IFIP 1985
- [Robinson 79] ROBINSON J.A.
Logic: Form and Function; the mechanization of Deductive reasoning.
Edinburgh University Press, 1979.
- [Ross 77] ROSS D. T.
Structured Analysis: a language for communicating ideas.
IEEE TOSE, vol SE 3, n° 1 1977
- [Rumbaugh 91] RUMBAUGH J. ; BLAHA M. ; PREMERLANI W. ; EDDY F. ; LORENSEN W.
Object-Oriented Modeling and Design
ISBN 0-13-630054-5 Prentice-Hall 1991
- [Sallé 87] SALLE P.
Langages d'Acteurs et Langages Objets: Le Langage Plasma.
Interkibernetik'87, Taragone.
- [Senteni...89] SENTENI A. ; SALLE P. ; LAPALME G.
Modelling Complex Behavior in Discrete Event Simulation with Actors.
Proceedings of SCSC'89 Summer Simulation Conference.
Austin (Tx) USA, July 1989
- [Senteni...90] SENTENI A. ; GIROUX S.
From Rock-Bottom to Metalevel Actors: Contribution to an Actor Programming
Methodology.
in [TOOLS'90] pp 167-181
- [Sernadas 89] SERNADAS A., FIADIEIRO J., SERNADAS C., EHRICH H.D.
The Basic Building Blocks of Information Systems.
IFIP TC 8 / WG 8.1. working conference. Namur, Belgium, 18-20 October, 1989.
North-Holland
- [Sibertin 85] SIBERTIN-BLANC C.
High-level Petri nets with Data Structure.
6th European Workshop on Petri Net and applications, Espoo (Finland), June 85
- [Sibertin 86] SIBERTIN-BLANC C.
Petri Nets with individuals and objects instead of tokens.
Internal report, extended version of [Sibertin 85].
- [Sibertin 87] SIBERTIN-BLANC C.
Petri Nets with Objects as a recursive programming language. Bigre+Globule 59, dec.
87 (in french).
- [Sibertin 91b] SIBERTIN-BLANC C.
Analysis of Petri Nets communicating through a Client-Server protocol. Internal report.
- [Sibertin 91] SIBERTIN-BLANC C.
Cooperative Objects for the Conceptual Modelling of Organizational Information
Systems.
Proc. of IFIP TC8 Working Conference on the Object Oriented Approach in
Information Systems, Quebec City, Canada October 1991.
F. van Assche et al. (Eds) Elsevier Science Publishers 1991
- [Sifakis 79] SIFAKIS J.
Use of Petri nets for performance evaluation.
3rd International symposium on modelling and performance evaluation of computer
systems. H. Beilner et E. Gelenbe Eds, North-Holland 1977

-
- [Souissi...89] SOUISSI Y., MEMMI G.
Composition of nets via a communication medium.
10th European Workshop on applications and theory of Petri Nets, Bonn, June 89.
- [Stefik...86] STEFIK M. ; BOBROW D.G.
Object-Oriented programming : themes and variations.
AI Magazine Vol.6 N°4 1986, pp.40-62.
- [Stroustrup...87] STROUSTRUP B.
What is "Object-Oriented Programming" ?
in [ECOOP 87] pp. 57-78
- [Tardieu...83] TARDIEU H. ; ROCHFELD A. ; COLETTI R.
La méthode Merise.
Edition des Organisations, Paris 1983
- [Taubner 87] TAUBNER D.
On the implementation of Petri Nets.
Proceedings of 8th European workshop on application and theory of P.N. ZARAGOZA 1987.
- [TECHLOG 89] TECHLOG S.A.
SEDRIC : Présentation générale, et : SEDRIC : Manuel de référence.
TECHLOG, Toulouse 1989
- [TOOLS'90] TECHNOLOGY OF OBJECT-ORIENTED LANGUAGES AND SYSTEMS
Proceedings of the Second International Conference TOOLS.
Bezivin, Meyer, Nerson Eds.
Angkor, Paris 1990
- [Tripathi...89] TRIPATHI A. ; BERGE E. ; AKSIT M.
An Implementation of the Object-oriented Concurrent Programming Language SINA.
Software Practice and Experience, Vol. 19, N° 13, pp. 235-256, March 1989
ISBN 0038-0644
- [TSI 85] Technique et Science Informatique
"Spécial Réseaux de Petri"
Vol. 4, n° 1 Janvier 1985
- [Valette 79] R. VALETTE
Analysis of Petri Nets by Stepwise Refinements.
Journal of computer and system science 18, 3; 1979
- [Valette 86] VALETTE R.
Nets in Production Systems.
in [LNCS 87] pp. 191-217
- [Valette...85] VALETTE R. ; THOMAS V. ; BACHMANN S.
SEDRIC un simulateur à événements discrets basé sur les réseaux de Petri.
RAIRO/APII, Vol.19, n° 5, pp.423-436
- [Valette...88a] VALETTE R, PALUDETTO M.
Approche Orientée Objet HOOD et réseaux de Petri pour la conception de logiciel temps réel.
Software Engineering and its Applications, Toulouse, dec. 1988.
- [Valette...88b] VALETTE R. ; PALUDETTO M. ; LABREUILLE B.P ; FARAIL P.
Approche orientée-objet HOOD et réseaux de Petri pour la conception de logiciel temps réel.
Proc. of 1st International Workshop on Software Engineering and its Applications
EC2 Toulouse, France, December 1988

-
- [Vielcanet 90] VIELCANET P.
HOOD Design Method and Control/Command techniques for the Development of Real-Time Software.
CISI Ingénierie - Aerospace Branch 1990
- [Ward...85] WARD P., MELLOR S.
Structured analysis for Real Time systems. Prentice Hall, New Jersey, 1985.
- [Wasserman 85] WASSERMAN A.
Extending State Transition Diagrams for the Specification of Human-Computer Interaction.
IEEE Transactions on Software Engineering, Vol. 11, N° 18
August 1985
- [Weinreb...80] WEINREB D. ; MOON D.
Flavors: Message Passing in the Lisp Machine.
AI Memo 602, MIT, Artificial Intelligence Laboratory.
November 1980
- [Winston 81] WINSTON P.H. ; HORN B.K.P
LISP
Addison-Wesley, 1981
- [Wirfs-Brock...90] WIRFS-BROCK R. ; WILKERSON B. ; WIENER L.
Designing Object-Oriented Software.
Prentice-Hall, 1990
- [Wirth 71] WIRTH N.
Program development by step-wise refinement.
Communications of the ACM vol. 14, N° 4, pp 221-227
April 1971
- [Yokote...87] YOKOTE Y. ; TOKORO M.
Concurrent Programming in Concurrent Smalltalk.
in [Yonezawa...87]
- [Yonezawa...86] YONEZAWA A. ; MATSUDA H. ; SHIBAYAMA E.
An Approach to Object-Oriented Concurrent Programming: A language ABCL.
in [BI-GL 48] pp. 125-134
- [Yonezawa...87] YONEZAWA A. ; TOKORO M. (eds.)
Concurrent Object-Oriented Programming.
MIT Press, Cambridge, Mass.; USA 1987

Table des définitions

Chapitre I : Modélisation par objets des systèmes concurrents

Définition I.1	- Objet et Classe d'Objets	10
Définition I.2	- Sous-type.....	16
Définition I.3	- Préservation de la sémantique	17
Définition I.4	- Moniteur.....	20
Définition I.5	- Conditions d'un moniteur	20
Définition I.6	- Axiome de préservation de l'ordre de transmission.....	32

Chapitre II : Les réseaux de Petri de Haut Niveau

Définition II.1	- Multiensemble.....	45
Définition II.2	- Addition de deux multiensembles	45
Définition II.3	- Multiplication d'un multiensemble par un scalaire	46
Définition II.4	- Ordre partiel sur les multiensembles	46
Définition II.5	- Suites finies d'éléments	46
Définition II.6	- Concaténation de n-uplets	46
Définition II.7	- Longueur d'un n-uplet	46
Définition II.8	- Multiensemble de suites finies	47
Définition II.9	- Prolongement canonique	47
Définition II.10	- Prolongement linéaire	47
Définition II.11	- Support d'un multiensemble de n-uplets.....	48
Définition II.12	- Domaine d'un type.....	48
Définition II.13	- Valeur d'une entité	49
Définition II.14	- Compatibilité des types	49
Définition II.15	- Réseau de Petri à Objets	49
Définition II.16	- Jetons simples et jetons typés.....	52
Définition II.17	- Marquage d'un RPO	52
Définition II.18	- Substitution des variables d'une transition	52
Définition II.19	- Franchissabilité d'une transition	52
Définition II.20	- Entités créées par un franchissement.....	54
Définition II.21	- Marquage atteint par un franchissement.....	54
Définition II.22	- Transitions concurremment franchissables	55
Définition II.23	- Pondération d'un réseau	56
Définition II.24	- Invariant algébrique	56
Définition II.25	- Fonction uniforme.....	56
Définition II.26	- Conflit structurel et conflit effectif.....	59
Définition II.27	- Registres d'un RPO	60
Définition II.28	- Règle d'émission	61
Définition II.29	- Franchissabilité d'un RdP à priorités	63

Définition II.30	- RPO à priorités.....	64
Définition II.31	- Arcs inhibiteurs ([Hack 75])	66
Définition II.32	- Arcs inhibiteurs	67
Définition II.33	- RPO à arcs inhibiteurs.....	67
Définition II.34	- Franchissabilité dans un RPO à arcs inhibiteurs	68
Définition II.35	- Critère de tri d'une place	70
Définition II.36	- Place triée.....	71
Définition II.37	- Places triées et franchissabilité.....	72

Chapitre III : Définition des classes d'Objets Coopératifs

Définition III.1	- Spécification d'une classe d'Objets Coopératifs	113
Définition III.2	- Règle d'évolution d'un ObCS	114
Définition III.3	- Relation d'utilisation	129

Chapitre IV : Systèmes d'Objets Coopératifs et organisation des classes

Définition IV.1	- Héritage et structure de données	142
Définition IV.2	- Héritage et services	142
Définition IV.3	- Héritage multiple et renommage des services	143
Définition IV.4	- Critère de non-duplication.....	154

Chapitre VI : Extension du formalisme

Définition VI.I	- Services Data-Flow	198
-----------------	----------------------------	-----

Chapitre IX : Analyse d'un modèle d'Objets Coopératifs

Définition IX.1	- Séquences franchissables de transitions	244
Définition IX.2	- Critère d'honnêteté	245
Définition IX.3	- Critère de courtoisie	245
Définition IX.4	- Critère de transparence.....	245
Définition IX.5	- Critère d'implémentation fidèle	246
Définition IX.6	- Critère d'héritage du comportement	247
Définition IX.7	- Compatibilité entre client et serveur	248

Table des figures

Chapitre I : Modélisation par objets des systèmes concurrents

Figure I.1	- Une File d'entiers en Eiffel.....	14
Figure I.2	- Les deux aspects d'un contrat.....	15
Figure I.3	- Histoire de la concurrence selon Lamport	20
Figure I.4	- Contre-exemple.....	26
Figure I.5	- Unification des concepts de processus et de classe.....	29
Figure I.6	- Dynamique des objets actifs HOOD	36
Figure I.7	- Syntaxe graphique des objets HOOD.....	39

Chapitre II : Les réseaux de Petri de Haut Niveau

Figure II.1	- Construction des substitutions par semi-unification.....	53
Figure II.2	- conflit de transitions.....	59
Figure II.3	- Indéterminisme lié aux valeurs des jetons.....	60
Figure II.4	- Place registre et notation abrégée.....	61
Figure II.5	- Règle d'émission et notation abrégée	62
Figure II.6	- Place à capacité et notation abrégée.....	63
Figure II.7	- Priorités par transitions et priorités par arcs.....	65
Figure II.8	- Priorités sur les arcs et place à capacité	65
Figure II.9	- Arc inhibiteur	66
Figure II.10	- combinaison d'arc inhibiteur et d'arc standard	67
Figure II.11	- Arc inhibiteur généralisé sans précondition	69
Figure II.12	- Arc inhibiteur généralisé avec précondition.....	69
Figure II.13	- Arc inhibiteur testant tous les jetons d'une place.....	70
Figure II.14	- Influence du tri sur la franchissabilité	72
Figure II.15	- Tri et unification	72
Figure II.16	- Un motif se comportant comme une file	74
Figure II.17	- Evolution des marquages pour le motif File.....	75
Figure II.18	- Un motif se comportant comme une Pile	76
Figure II.19	- Evolution des marquages pour le motif Pile.....	77
Figure II.20	- Motif se comportant comme une place triée (version 1)	78
Figure II.21	- Evolution des marquages pour le motif Place triée	79
Figure II.22	- Motif se comportant comme une place triée (version 2)	79
Figure II.23	- émulation d'une transition temporisée	82
Figure II.24	- émulation des réseaux à arcs temporels.....	83

Chapitre III : Définition des classes d'Objets Coopératifs

Figure III.1	- Equation fondamentale du modèle	93
Figure III.2	- Le formalisme des Objets Coopératifs et ses formalismes de	
référence	93	
Figure III.3	- Syntaxe de l'interface d'une classe.....	101
Figure III.4	- Syntaxe graphique de la pragmatique d'une classe.....	104
Figure III.5	- BNF de la définition du marquage initial d'une place	106
Figure III.6	- Spécification de la classe Fichier	108
Figure III.7	- Spécification de la classe Fichier_protégé	110
Figure III.8	- Services d'initialisation et de mise à jour	113
Figure III.9	- Exemple de transition d'invocation	120
Figure III.10	- Transition d'invocation et d'accès.....	120
Figure III.11	- Implémentation de la classe Fichier	123
Figure III.12	- Implémentation de la classe Fichier_protégé	124
Figure III.13	- OpCS de l'opération Ouvrir_en_écriture.....	125
Figure III.14	- OpCS de l'opération Lire.....	126
Figure III.15	- Les deux côtés d'une invocation.....	128
Figure III.16	- Sémantique intuitive d'une invocation.....	129

Chapitre IV : Systèmes d'Objets Coopératifs et organisation des classes

Figure IV.1	- Syntaxe de définition d'un système d'Objets Coopératifs.....	137
Figure IV.2	- Univers, Système d'Objets et Environnement	140
Figure IV.3	- Spécification de la classe Composite_Exemple	148
Figure IV.4	- Définition textuelle de l'implémentation de la classe	
Composite_Exemple	149	
Figure IV.5	- Définition graphique de l'implémentation de la classe	
Composite_Exemple	150	
Figure IV.6	- Configuration d'une table de philosophes	151
Figure IV.7	- Implémentation de la classe Philosophe.....	153
Figure IV.8	- Spécification de la classe Table	155
Figure IV.9	- Définition textuelle de l'implémentation de la Table.....	155
Figure IV.10	- Définition graphique de l'implémentation de la Table	156

Chapitre V : Sémantique formelle d'un système d'Objets Coopératifs

Figure V.1	- Génération d'identificateurs	158
Figure V.2	- La classe Pelient.....	159
Figure V.3	- La classe ServeurG.....	160
Figure V.4	- La classe ServeurS	160

Figure V.5	- Définition du système-exemple	161
Figure V.6	- ObCS de Pclient (étape 1)	164
Figure V.7	- ObCS de ServeurG (étape 2)	165
Figure V.8	- ObCS d'extension de Pclient et ServeurG (étape 3)	168
Figure V.9	- Définition de la classe Cclient	168
Figure V.10	- ObCS d'extension de Cclient1	169
Figure V.11	- Marquage initial du système-exemple	170
Figure V.12	- accès itératif aux attributs publics	172
Figure V.13	- Implantation des accès aux attributs dans le WSCS	173
Figure V.14	- WSCS du système-exemple	174
Figure V.15	- Instanciation et invocation d'une classe composite	177
Figure V.16	- Réseau de traduction pour l'instanciation d'une classe composite	178
Figure V.17	- Réseau de traduction pour l'invocation d'une instance composite	179
Figure V.18	- Définition du système (version 1)	180
Figure V.19	- Définition du système (version 2)	181
Figure V.20	- WSCS de la table de philosophes	182
Figure V.21	- Réseau simplifié de la table de philosophes	184

Chapitre VI : Extension du formalisme

Figure VI.1	- Syntaxe graphique d'une invocation non confirmée	188
Figure VI.2	- Sémantique d'une invocation non confirmée	189
Figure VI.3	- Syntaxe d'une invocation gardée	190
Figure VI.4	- Sémantique d'une invocation gardée	191
Figure VI.5	- Exemples de la syntaxe d'invocations gardées	194
Figure VI.6	- Invocation d'un service priorisé	195
Figure VI.7	- WSCS d'un service LIFO	196
Figure VI.8	- Spécification et sémantique d'un service de mise à jour	197
Figure VI.9	- Classe Cserveur, publiant un data-flow	198
Figure VI.10	- ObCS d'extension de Cserveur	199
Figure VI.11	- WSCS généré par l'invocation d'un data-flow	200

Chapitre VII : Etudes de cas

Figure VII.1	- Plan général de l'atelier	206
Figure VII.2	- Architecture d'une cellule de production	208
Figure VII.3	- Fonctionnement d'une cellule de production	209
Figure VII.4	- Paramètres des composants de l'atelier	210
Figure VII.5	- Liste des opérations par poste	210
Figure VII.6	- Graphes d'héritage et de composition	212
Figure VII.7	- La classe Palette	213
Figure VII.8	- Classe composite Atelier	214
Figure VII.9	- Spécification de la classe Transport	215
Figure VII.10	- Implémentation de la classe Transport	216

Figure VII.11	- Spécification de la classe Dérivation	217
Figure VII.12	- Implémentation de la classe Dérivation	218
Figure VII.13	- Spécification de la classe composite Ilot.....	219
Figure VII.14	- Implémentation de la classe composite Ilot.....	219
Figure VII.15	- Spécification de la classe Transport_double	220
Figure VII.16	- Implémentation de la classe Transport_double	221
Figure VII.17	- Spécification de la classe Entrée_ilot.....	222
Figure VII.19	- Spécification de la classe Cellule	224
Figure VII.20	- Le modèle Seeheim	227
Figure VII.21	- Structure de contrôle d'une application interactive conventionnelle	229
Figure VII.22	- Structure de contrôle d'une application dirigée par événements	230
Figure VII.23	- Représentation externe de l'éditeur de n-uplets	233
Figure VII.24	- Spécification de la classe n-uplet	234
Figure VII.25	- L'éditeur de n-uplets avec un message d'erreur modal	235
Figure VII.26	- Implémentation de la classe Editeur.....	236

Chapitre IX : Analyse d'un modèle d'Objets Coopératifs

Figure IX.1	- Serveur sournois et serveur transparent.....	246
-------------	--	-----

